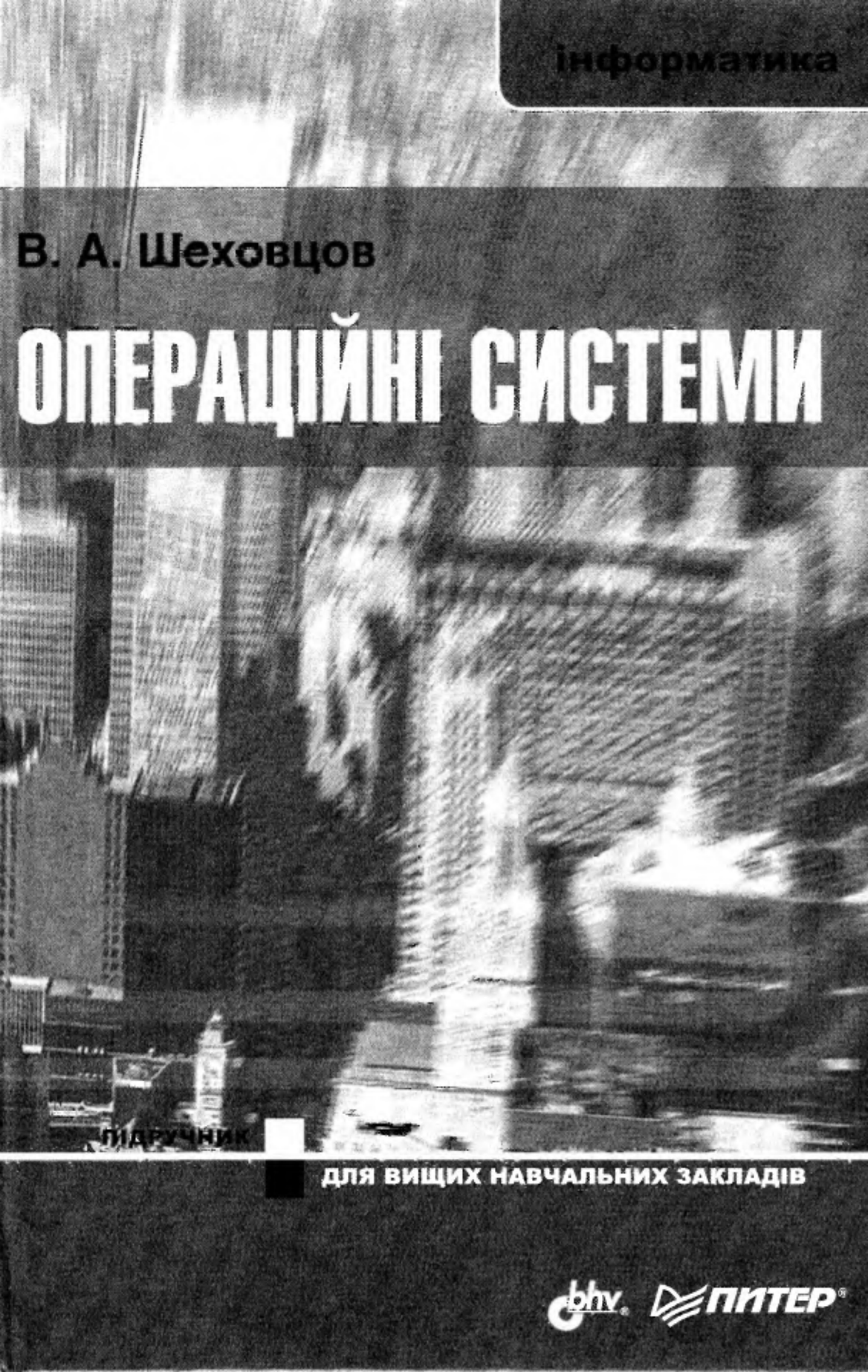
інформатика
В. А. Шеховцов
ОПЕРАЦІЙНІ СИСТЕМИ
ПІДРУЧНИК
ДЛЯ ВИЩИХ НАВЧАЛЬНИХ ЗАКЛАДІВ
bhv
ПИТЕР

Навчальне видання

Шеховцов Володимир Анатолійович
ОПЕРАЦІЙНІ СИСТЕМИ

Підручник

Керівник проекту В. П. Пасько
Редактор С. Г. Єзерницька
Коректор Н. М. Тонгоног
Комп'ютерна верстка Д. С. Тріщенкова

ТОВ «Видавнича група BHV»
Свідоцтво про внесення до Державного реєстру
суб'єктів видавничої справи України
серія ДК №175 від 13.09.2000 р.

Підписано до друку 09.08.05. Формат $70\times100^{1}/_{16}$.
Папір офсетний. Гарнітура Petersburg. Друк офсетний.
Ум. друк. арк. 46,44. Обл.-вид. арк. 43,55.
Наклад 3000 прим. Зам. № 30144.

Виготовлено в ТОВ «Освітня книга»,
м. Київ, вул. Орловська, 2/7, оф. 6.
Свідоцтво про внесення до Державного реєстру
суб'єктів видавничої справи України
серія ДК № 2245 від 26.07.2005 р.

В. А. ШЕХОВЦОВ

ОПЕРАЦІЙНІ СИСТЕМИ

Затверджено Міністерством освіти і науки України як підручник для студентів вищих навчальних закладів, які навчаються за напрямами «Комп'ютерні науки», «Комп'ютеризовані системи, автоматика і управління», «Комп'ютерна інженерія», «Прикладна математика»

Серія «ІНФОРМАТИКА»
За загальною редакцією академіка НАН України
М. З. Згуровського

Київ
Видавнича група BHV
2005

ББК 32.973-018.2я73
Ш-54
УДК 004.451(075.8)

Рецензенти: Г. М. Жолткевич, доктор технічних наук, професор, завідувач кафедри теоретичної і прикладної інформатики, директор Центра комп'ютерних технологій Харківського національного університету ім. В. Н. Каразіна;

М. О. Сидоров, доктор технічних наук, професор, декан факультету комп'ютерних наук, завідувач кафедри інженерії програмного забезпечення Національного авіаційного університету.

Гриф надано Міністерством освіти і науки України, лист № 14/18.2-488 від 02.03.2005 р.

Шеховцов В. А.

Ш-54 Операційні системи. — К.: Видавнича група BHV, 2005. — 576 с.: іл.
ISBN 966-552-157-8

Підручник містить повний і систематизований виклад фундаментальних концепцій сучасних операційних систем. Розглядаються основні функції операційних систем: керування процесами, пам'яттю, файлові системи, засоби вводу-виводу, мережні засоби, забезпечення безпеки тощо. Всі теоретичні положення підкріплюються прикладами, що розкривають особливості організації операційних систем UNIX/Linux та Windows XP.

Характерною рисою даного підручника є те, що він містить детальний опис програмних інтерфейсів, необхідних прикладним програмам для доступу до засобів операційних систем. У підручнику вдало поєднано виклад теоретичних концепцій побудови операційних систем і опис особливостей практичної реалізації з прикладами коду. Розглядається специфіка використання системних викликів UNIX/Linux і функцій Win32 API, наводяться численні приклади програмного коду.

Книжка призначена для студентів, які навчаються за напрямом «Комп'ютерні науки» і вивчають сучасні інформаційні технології в рамках дисципліни «Системне програмування і операційні системи», а також для викладачів зазначеної дисципліни. Книжка також може бути корисна професійним програмістам і системним адміністраторам.

ББК 32.973-018.2я73

Серія підручників «Інформатика» виходить у світ за підтримки видавництв «Издательский дом "Питер"», м. Санкт-Петербург та «Освітня книга», м. Київ.

ISBN 966-552-157-8

ЗМІСТ

Світлої пам'яті моєї мами
Валентини Анатоліївни Шеховцової

Передмова

Підручник містить повний і систематизований виклад фундаментальних концепцій і практичних рішень, що лежать в основі сучасних операційних систем. Головною характеристикою підручника є його практична спрямованість. Зокрема, для вивчення були ретельно відібрані теоретичні концепції, які застосовуються у сучасних операційних системах, при цьому кожна з них підкріплюється практичними прикладами, що розкривають особливості організації UNIX/Linux і Windows XP/Windows Server 2003.

Книга складається з 20 розділів. У перших двох розділах наведено базові означення курсу, розглянуто основні функції сучасних операційних систем, введено поняття архітектури операційної системи, описано архітектурні особливості Linux і Windows XP.

Наступні п'ять розділів, із третього по сьомий, присвячено основним «активним» компонентам операційних систем – процесам і потокам. Далі, у розділах з восьмого по десятий, розглянуто керування пам'яттю. Наступні три розділи, з одинадцятого по тринадцятий, присвячені файловим системам.

В інших розділах книги розглянуто додаткові можливості й альтернативні архітектури операційних систем, а саме, організацію і завантаження у пам'ять програмного коду, керування пристроями введення-виведення, особливості організації мережних засобів операційних систем, взаємодію з користувачем, засоби захисту інформації, процес завантаження операційних систем та реалізацію багатопроцесорних і розподілених операційних систем.

Інформація про Linux базується на останніх версіях ядра (2.4–2.6), в роботі над прикладами переважно використана система Red Hat Linux 7.2. Більшість описаних системних викликів відповідає стандарту POSIX, тому цей матеріал також може бути використаний під час вивчення інших версій UNIX.

Матеріал щодо Windows XP/Windows Server 2003 загалом залишається справедливим і для попередніх версій системи (у першу чергу Windows 2000), при цьому описано всі відмінності між системами. Опис застарілих систем лінії Consumer Windows (Windows 95/98/Me) не наводиться.

Цей підручник відрізняється від інших тим, що в ньому наведено детальний опис програмних інтерфейсів, необхідних для одержання доступу до відповідних засобів операційних систем із прикладних програм. Описано особливості використання більш ніж 250 системних викликів UNIX/Linux і функцій Win32 API, наведено численні приклади програмного коду, що застосовує ці виклики й функції.

Програмний код використовує тільки стандартні засоби ANSI C та програмних інтерфейсів операційних систем; для компіляції цього коду може бути використаний будь-який засіб розробки, що підтримує відповідні стандарти (для розробки коду під Linux автор застосовував gcc, зручним безкоштовним засобом розробки під Windows XP є Microsoft Visual C++ Toolkit у поєднанні з Microsoft Platform SDK).

Підручник призначено для студентів денної форми навчання під час вивчення ними курсу «Системне програмування й операційні системи», передбаченого програмою підготовки бакалаврів напряму «Комп'ютерні науки», а також для студентів, які навчаються за напрямами «Комп'ютерна інженерія», «Прикладна математика» та «Комп'ютеризовані системи, автоматика і управління». Практична спрямованість книги робить її корисною і для фахівців. Наприклад, нею можуть користуватися розробники програмного забезпечення та системні адміністратори для підвищення своєї кваліфікації в галузі операційних систем.

Створення книги було б неможливим без допомоги багатьох людей. Автор дякує за підтримку завідувачу кафедри автоматизованих систем управління Національного технічного університету «Харківський політехнічний інститут» Михайлу Дмитровичу Годлевському, співробітникам кафедри. Цінні поради під час роботи над матеріалом книги були отримані від співробітників фірми Ratmir Labs (Харків) Андрія Кудряшова і Костянтина Киренка, а також від професора кафедри вищої математики й інформатики Харківського національного університету ім. В. Н. Каразіна Григорія Миколайовича Жолткевича.

Автор також хотів би подякувати співробітникам видавництва BHV, зокрема Олександрові Полякову та Вікторові Лебедку, за їхню професійну допомогу в роботі над книгою та підготовці рукопису до друку.

І головне — автор вдячний своїй родині, насамперед дружині Саші, за героїчне терпіння та стійкість, які були потрібні протягом всього часу, коли замислювалася, створювалася і перероблялася ця книга.

На сайтах http://www.osvita.info та http://www.bhv.kiev.ua містяться добірка вправ за матеріалами різних розділів, регулярно оновлюваний список інтернет-ресурсів за темою книги, архів програмних кодів та різноманітні додаткові матеріали.

Від видавництва

Ваші зауваження, пропозиції та запитання надсилайте за адресою електронної пошти pg@bhv.kiev.ua та за адресою http://www.osvita.info. На цьому ж сайті ви познайомитеся з детальною інформацією про інші видання серії «Інформатика».

Інформацію про книжки Видавничої групи BHV ви можете знайти на сайті http://www.bhv.kiev.ua.

Розділ 1

Основні концепції операційних систем

- Поняття операційної системи та її призначення
- Історія розвитку операційних систем
- Класифікація операційних систем
- Основні функції операційної системи

У цьому розділі буде дано поняття операційної системи, описано призначення різних операційних систем, виділено їхні базові функції та служби. Розділ також містить короткий огляд історії розвитку операційних систем.

1.1. Поняття операційної системи, її призначення та функції

1.1.1. Поняття операційної системи

Причиною появи операційних систем була необхідність створення зручних у використанні комп'ютерних систем (під *комп'ютерною системою* будемо розуміти сукупність апаратного і програмного забезпечення комп'ютера). Комп'ютерні системи від самого початку розроблялися для розв'язання практичних задач користувачів. Оскільки робити це за допомогою лише апаратного забезпечення виявилося складно, були створені прикладні програми. Для таких програм знадобилися загальні операції керування апаратним забезпеченням, розподілу апаратних ресурсів тощо. Ці операції згрупували в рамках окремого рівня програмного забезпечення, який і стали називати операційною системою.

Далі можливості операційних систем вийшли далеко за межі базового набору операцій, необхідних прикладним програмам, але проміжне становище таких систем між прикладними програмами й апаратним забезпеченням залишилося незмінним.

Можна дати таке означення операційної системи.

Операційна система (ОС) – це програмне забезпечення, що реалізує зв'язок між прикладними програмами й апаратними засобами комп'ютера.

1.1.2. Призначення операційної системи

Операційні системи забезпечують, по-перше, зручність використання комп'ютерної системи, по-друге, ефективність і надійність її роботи.

Перша функція властива ОС як розширеній машині, друга – ОС як розподілювача апаратних ресурсів.

1.1.3. Операційна система як розширена машина

За допомогою операційної системи у прикладного програміста (а через його програми і в користувача) має створюватися враження, що він працює з розширеною машиною [4, 29, 41, 44].

Апаратне забезпечення комп'ютера недостатньо пристосоване до безпосереднього використання у програмах. Наприклад, якщо розглянути роботу із пристроями введення-виведення на рівні команд відповідних контролерів, то можна побачити, що набір таких команд обмежений, а для багатьох пристроїв – примітивний (є навіть вислів: «апаратне забезпечення потворне»). Операційна система приховує такий *інтерфейс апаратного забезпечення*, замість нього програмістові пропонують *інтерфейс прикладного програмування* (рис. 1.1), що використовує поняття вищого рівня (їх називають абстракціями).

Наприклад, при роботі з диском типовою абстракцією є файл. Працювати з файлами простіше, ніж безпосередньо з контролером диска (не потрібно враховувати переміщення головок дисковода, запускати й зупиняти мотор тощо), внаслідок цього програміст може зосередитися на суті свого прикладного завдання. За взаємодію з контролером диска відповідає операційна система.

Виділення абстракцій дає змогу досягти того, що код ОС і прикладних програм не потребуватиме зміни при переході на нове апаратне забезпечення. Наприклад, якщо встановити на комп'ютері дисковий пристрій нового типу (за умови, що він підтримується ОС), всі його особливості будуть враховані на рівні ОС, а прикладні програми продовжуватимуть використовувати файли, як і раніше. Така характеристика системи називається *апаратною незалежністю*. Можна сказати, що ОС надають апаратно-незалежне середовище для виконання прикладних програм.

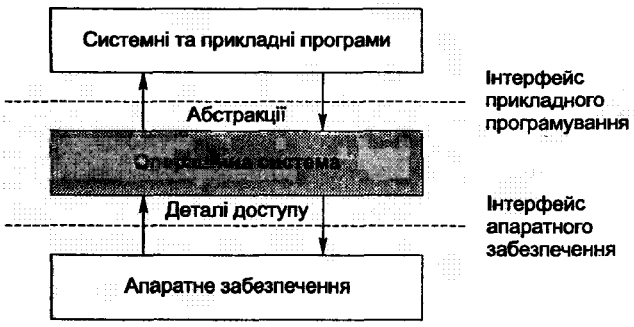

Рис. 1.1. Взаємодія ОС із апаратним забезпеченням і застосуваннями

1.1.4. Операційна система як розподілювач ресурсів

Операційна система має ефективно розподіляти ресурси. Під ресурсами розуміють процесорний час, дисковий простір, пам'ять, засоби доступу до зовнішніх пристроїв. Операційна система виступає в ролі менеджера цих ресурсів і надає їх прикладним програмам на вимогу.

Розрізняють два основні види розподілу ресурсів. У разі *просторового розподілу* ресурс доступний декільком споживачам одночасно, при цьому кожен із них може користуватися частиною ресурсу (так розподіляється пам'ять). У разі *часового розподілу* система ставить споживачів у чергу і згідно з нею надає їм змогу користуватися всім ресурсом обмежений час (так розподіляється процесор в однопроцесорних системах).

При розподілі ресурсів ОС розв'язує можливі конфлікти, запобігає несанкціонованому доступу програм до тих ресурсів, на які вони не мають прав, забезпечує ефективну роботу комп'ютерної системи.

1.2. Історія розвитку операційних систем

Перші операційні системи з'явилися в 50-ті роки і були *системами пакетної обробки*. Такі системи забезпечували послідовне виконання програм у пакетному режимі (без можливості взаємодії з користувачем). У певний момент часу в пам'яті могла перебувати тільки одна програма (системи були однозадачними), усі програми виконувалися на процесорі від початку до кінця. За такої ситуації ОС розглядали просто як набір стандартних служб, необхідних прикладним програмам і користувачам.

Наступним етапом стала підтримка багатозадачності. У багатозадачних системах у пам'ять комп'ютера стали завантажувати кілька програм, які виконувалися на процесорі навперемінно. При цьому розвивалися два напрями: *багатозадачна пакетна обробка* і *розподіл часу*. У багатозадачній пакетній обробці завантажені програми, як і раніше, виконувалися в пакетному режимі. У режимі розподілу часу із системою могли працювати одночасно кілька користувачів, кожному з яких надавався діалоговий термінал (пристрій, що складається із клавіатури і дисплея).

Підтримка багатозадачності потребувала реалізації в ОС засобів координації задач. Можна виділити три складові частини такої координації.

1. Захист критичних даних задачі від випадкового або навмисного доступу інших задач.
2. Забезпечення обміну даними між задачами.
3. Надання задачам справедливої частки ресурсів (пам'яті, процесора, дискового простору тощо).

Ще одним етапом стала поява *ОС персональних комп'ютерів*. Спочатку ці системи, як і ОС першого етапу, були однозадачними й надавали базовий набір стандартних служб (на цьому етапі важливим було впровадження графічного інтерфейсу користувача). Подальший розвиток апаратного забезпечення дав змогу використати в таких системах засоби, розроблені для більших систем, насамперед багатозадачність і, як наслідок, координацію задач.

Є правило розвитку ОС для конкретної апаратної платформи: для більшості нових апаратних платформ ОС спочатку створюють як базовий набір стандартних служб, координацію задач реалізують у ній пізніше. Зазначимо, що це правило вірне, якщо апаратна платформа дозволяє реалізувати багатозадачний режим.

Багато сучасних ОС спочатку розроблялися для персональних комп'ютерів або були перенесені на них з інших апаратних платформ. Основну увагу в цій книзі буде приділено двом групам операційних систем: UNIX-сумісним системам, насамперед Linux, та серії Windows NT/2000/XP фірми Microsoft (далі називатимемо ці системи лінією Windows XP).

Більш докладно з історичним нарисом розвитку операційних систем можна ознайомитися в літературі [19, 29, 44, 104].

1.3. Класифікація сучасних операційних систем

Розглянемо класифікацію сучасних операційних систем залежно від області їхнього застосування.

Насамперед відзначимо *ОС великих ЕОМ* (мейнфреймів). Основною характеристикою апаратного забезпечення, для якого їх розробляють, є продуктивність введення-виведення: великі ЕОМ оснащують значною кількістю периферійних пристроїв (дисків, терміналів, принтерів тощо). Такі комп'ютерні системи використовують для надійної обробки значних обсягів даних, при цьому ОС має ефективно підтримувати цю обробку (в пакетному режимі або в режимі розподілу часу). Прикладом ОС такого класу може бути OS/390 фірми IBM.

До наступної категорії можна віднести *серверні ОС*. Головна характеристика таких ОС – здатність обслуговувати велику кількість запитів користувачів до спільно використовуваних ресурсів. Важливу роль для них відіграє мережна підтримка. Є спеціалізовані серверні ОС, з яких виключені елементи, не пов'язані з виконанням їхніх основних функцій (наприклад, підтримка застосувань користувача). Нині для реалізації серверів частіше застосовують універсальні ОС (UNIX або системи лінії Windows XP).

Наймасовіша категорія – *персональні ОС*. Деякі ОС цієї категорії розробляли з розрахунком на непрофесійного користувача (лінія Windows 95/98/Me фірми Microsoft, яку далі називатимемо Consumer Windows), інші є спрощеними версіями універсальних ОС. Особлива увага в персональних ОС приділяється підтримці графічного інтерфейсу користувача і мультимедіа-технологій.

Виділяють також *ОС реального часу*. У такій системі кожна операція має бути гарантовано виконана в заданому часовому діапазоні. ОС реального часу можуть керувати польотом космічного корабля, технологічним процесом або демонстрацією відеороликів. Існують спеціалізовані ОС реального часу, такі як QNX [1, 7, 11] і VxWorks.

Ще однією категорією є *вбудовані ОС*. До них належать керуючі програми для різноманітних мікропроцесорних систем, які використовують у військовій техніці, системах побутової електроніки, смарт-картах та інших пристроях. До таких систем ставлять особливі вимоги: розміщення в малому обсязі пам'яті, підтримка спеціалізованих засобів введення-виведення, можливість прошивання в постій-

ному запам'ятовувальному пристрої. Часто вбудовані ОС розробляються під конкретний пристрій; до універсальних систем належать Embedded Linux [68] і Windows CE [49].

1.4. Функціональні компоненти операційних систем

Операційну систему можна розглядати як сукупність функціональних компонентів, кожен з яких відповідає за реалізацію певної функції системи. У цьому розділі описані найважливіші функції сучасних ОС і компоненти, що їх реалізують.

Спосіб побудови системи зі складових частин та їхній взаємозв'язок визначає архітектура операційної системи. Підходи до реалізації архітектури ОС будуть розглянуті в розділі 2.

1.4.1. Керування процесами й потоками

Як ми вже згадували, однією з найважливіших функцій ОС є виконання прикладних програм. Код і дані прикладних програм зберігаються в комп'ютерній системі на диску в спеціальних виконуваних файлах. Після того як користувач або ОС вирішать запустити на виконання такий файл, у системі буде створено базову одиницю обчислювальної роботи, що називається *процесом* (process).

Можна дати таке означення: процес — це програма під час її виконання.

Операційна система розподіляє ресурси між процесами. До таких ресурсів належать процесорний час, пам'ять, пристрої введення-виведення, дисковий простір у вигляді файлів. При розподілі пам'яті з кожним процесом пов'язується його *адресний простір* — набір адрес пам'яті, до яких йому дозволено доступ. В адресному просторі зберігаються код і дані процесу. При розподілі дискового простору для кожного процесу формується список відкритих файлів, аналогічним чином розподіляють пристрої введення-виведення.

Процеси забезпечують захист ресурсів, якими вони володіють. Наприклад, до адресного простору процесу неможливо безпосередньо звернутися з інших процесів (він є захищеним), а при роботі з файлами може бути задано режим, що забороняє доступ до файла всім процесам, крім поточного.

Розподіл процесорного часу між процесами необхідний через те, що процесор виконує інструкції одну за одною (тобто в конкретний момент часу на ньому може фізично виконуватися тільки один процес), а для користувача процеси мають виглядати як послідовності інструкцій, виконувані паралельно. Щоб домогтися такого ефекту, ОС надає процесор кожному процесу на деякий короткий час, після чого перемикає процесор на інший процес; при цьому виконання процесів відновлюється з того місця, де їх було перервано. У *багатопроцесорній системі* процеси можуть виконуватися паралельно на різних процесорах.

Сучасні ОС крім багатозадачності можуть підтримувати *багатопотоковість* (multithreading), яка передбачає в рамках процесу наявність кількох послідовностей інструкцій (*потоків*, threads), які для користувача виконуються паралельно, подібно до самих процесів в ОС. На відміну від процесів потоки не забезпечують

захисту ресурсів (наприклад, вони спільно використовують адресний простір свого процесу).

1.4.2. Керування пам'яттю

Під час виконання програмного коду процесор бере інструкції й дані з оперативної (основної) пам'яті комп'ютера. При цьому така пам'ять відображається у вигляді масиву байтів, кожен з яких має адресу.

Як уже згадувалося, основна пам'ять є одним із видів ресурсів, розподілюваних між процесами. ОС відповідає за виділення пам'яті під захищений адресний простір процесу і за вивільнення пам'яті після того, як виконання процесу буде завершено. Обсяг пам'яті, доступний процесу, може змінюватися в ході виконання, у цьому разі говорять про динамічний розподіл пам'яті.

ОС повинна забезпечувати можливість виконання програм, які окремо або в сукупності перевищують за обсягом доступну основну пам'ять. Для цього в ній має бути реалізована технологія *віртуальної пам'яті*. Така технологія дає можливість розміщувати в основній пам'яті тільки ті інструкції й дані процесу, які потрібні в поточний момент часу, при цьому вміст іншої частини адресного простору зберігається на диску.

1.4.3. Керування введенням-виведенням

Операційна система відповідає за керування пристроями введення-виведення, підключеними до комп'ютера. Підтримка таких пристроїв в ОС звичайно здійснюється на двох рівнях. До першого, нижчого, рівня належать *драйвери пристроїв* — програмні модулі, які керують пристроями конкретного типу з урахуванням усіх їхніх особливостей. До другого рівня належить *універсальний інтерфейс введення-виведення*, зручний для використання у прикладних програмах.

ОС має реалізовувати загальний інтерфейс драйверів введення-виведення, через який вони взаємодіють з іншими компонентами системи. Такий інтерфейс дає змогу спростити додавання в систему драйверів для нових пристроїв.

Сучасні ОС надають великий вибір готових драйверів для конкретних периферійних пристроїв. Що більше пристроїв підтримує ОС, то більше в неї шансів на практичне використання.

1.4.4. Керування файлами та файлові системи

Для користувачів ОС і прикладних програмістів дисковий простір надається у вигляді сукупності *файлів*, організованих у *файлову систему*.

Файл — це набір даних у файловій системі, доступ до якого здійснюється за іменем. Термін «файлова система» може вживатися для двох понять: принципу організації даних у вигляді файлів і конкретного набору даних (зазвичай відповідної частини диска), організованих відповідно до такого принципу. У рамках ОС може бути реалізована одночасна підтримка декількох файлових систем.

Файлові системи розглядають на логічному і фізичному рівнях. Логічний рівень визначає зовнішнє подання системи як сукупності файлів (які звичайно перебувають у каталогах), а також виконання операцій над файлами і каталогами

(створення, вилучення тощо). Фізичний рівень визначає принципи розміщення структур даних файлової системи на диску або іншому пристрої.

1.4.5. Мережна підтримка

Мережні системи

Сучасні операційні системи пристосовані до роботи в мережі, їх називають *мережними операційними системами* [29]. Засоби мережної підтримки дають ОС можливість:

- надавати локальні ресурси (дисковий простір, принтери тощо) у загальне користування через мережу, тобто функціонувати як сервер;
- звертатися до ресурсів інших комп'ютерів через мережу, тобто функціонувати як клієнт.

Реалізація функціональності сервера і клієнта базується на *транспортних засобах*, відповідальних за передачу даних між комп'ютерами відповідно до правил, обумовлених мережними протоколами.

Розподілені системи

Мережні ОС не приховують від користувача наявність мережі, мережна підтримка в них не визначає структуру системи, а збагачує її додатковими можливостями. Є також *розподілені ОС* [6, 45], які дають змогу об'єднати ресурси декількох комп'ютерів у *розподілену систему*. Вона виглядає для користувача як один комп'ютер з декількома процесорами, що працюють паралельно. Розподілені та багатопроцесорні системи є двома основними категоріями ОС, які використовують декілька процесорів. Вони мають багато спільного й будуть розглянуті в розділі 20.

1.4.6. Безпека даних

Під безпекою даних в ОС розуміють забезпечення надійності системи (захисту даних від втрати у разі збоїв) і захист даних від несанкціонованого доступу (випадкового чи навмисного).

Для захисту від несанкціонованого доступу ОС має забезпечувати наявність засобів *аутентифікації* користувачів (такі засоби дають змогу з'ясувати, чи є користувач тим, за кого себе видає; зазвичай для цього використовують систему паролів) та їхньої *авторизації* (дозволяють перевірити права користувача, що пройшов аутентифікацію, на виконання певної операції).

1.4.7. Інтерфейс користувача

Розрізняють два типи засобів взаємодії користувача з ОС: *командний інтерпретатор* (shell) і *графічний інтерфейс користувача* (GUI).

Командний інтерпретатор дає змогу користувачам взаємодіяти з ОС, використовуючи спеціальну командну мову (інтерактивно або через запуск на виконання командних файлів). Команди такої мови змушують ОС виконувати певні дії (наприклад, запускати програми, працювати із файлами).

Графічний інтерфейс користувача надає йому можливість взаємодіяти з ОС, відкриваючи вікна і виконуючи команди за допомогою меню або кнопок. Підходи до реалізації графічного інтерфейсу доволі різноманітні: наприклад, у Windows-системах засоби його підтримки вбудовані в систему, а в UNIX вони є зовнішніми для системи і спираються на стандартні засоби керування введенням-виведенням.

Висновки

- Операційна система – це рівень програмного забезпечення, що перебуває між рівнями прикладних програм й апаратного забезпечення комп'ютера. Головне її призначення – зробити використання комп'ютерної системи простішим і підвищити ефективність її роботи.
- До основних функціональних компонентів ОС належать: керування процесами, керування пам'яттю, керування введенням-виведенням, керування файлами і підтримка файлових систем, мережна підтримка, забезпечення захисту даних, реалізація інтерфейсу користувача.

Контрольні запитання та завдання

1. Які основні функції операційної системи? Чи немає між ними протиріч?
2. Наведіть кілька прикладів просторового і часового розподілу ресурсів комп'ютера. Від чого залежить вибір того чи іншого методу розподілу?
3. У чому полягає основна відмінність багатозадачних пакетних систем від систем з розподілом часу? Як можна в рамках однієї системи об'єднати можливості обох зазначених систем?
4. Чому більшість вбудованих систем розроблено як системи реального часу? Наведіть приклади вбудованих систем, для яких підтримка режиму реального часу не є обов'язковою.
5. Що спільного й у чому відмінності між мережною і розподіленою операційними системами? Яка з них складніша в реалізації і чому?

Розділ 2

Архітектура операційних систем

- Означення архітектури операційних систем
- Ядро системи та системне програмне забезпечення
- Підходи до реалізації архітектури операційних систем
- Взаємодія операційної системи та апаратного забезпечення
- Взаємодія операційної системи та прикладних програм
- Архітектура UNIX і Linux
- Архітектура Windows XP

Операційну систему можна розглядати як сукупність компонентів, кожен з яких відповідає за певні функції. Набір таких компонентів і порядок їхньої взаємодії один з одним та із зовнішнім середовищем визначається *архітектурою операційної системи.*

У цьому розділі ми ознайомимося з основними поняттями архітектури операційних систем, підходами до її реалізації, особливостями взаємодії ОС із зовнішнім середовищем. Реалізацію архітектури буде розглянуто на прикладах UNIX, Linux і Windows XP.

2.1. Базові поняття архітектури операційних систем

2.1.1. Механізми і політика

В ОС насамперед необхідно виділити набір фундаментальних можливостей, які надають її компоненти; ці базові можливості становлять *механізм* (mechanism). З іншого боку, необхідно приймати рішення щодо використання зазначених можливостей; такі рішення визначають *політику* (policy). Отже, механізм показує, що реалізовано компонентом, а політика — як це можна використати.

Коли за реалізацію механізму і політики відповідають різні компоненти (механізм відокремлений від політики), спрощується розробка системи і підвищується її гнучкість. Компонентам, що реалізують механізм, не повинна бути доступна інформація про причини та цілі його застосування; усе, що потрібно від них, — це

виконувати призначену їм роботу. Для таких компонентів використовують термін «вільні від політики» (policy-free). Компоненти, відповідальні за політику, мають оперувати вільними від неї компонентами як будівельними блоками, для них недоступна інформація про деталі реалізації механізму.

Прикладом відокремлення механізму від політики є керування введенням-виведенням. Базові механізми доступу до периферійних пристроїв реалізують драйвери. Політику використання цих механізмів задає програмне забезпечення, що здійснює введення-виведення. Докладніше це питання буде розглянуте у розділі 15.

2.1.2. Ядро системи. Привілейований режим і режим користувача

Базові компоненти ОС, які відповідають за найважливіші її функції, зазвичай перебувають у пам'яті постійно і виконуються у привілейованому режимі, називають *ядром операційної системи* (operating system kernel).

Існуючі на сьогодні підходи до проектування архітектури ОС по-різному визначають функціональність ядра. До найважливіших функцій ОС, виконання яких звичайно покладають на ядро, належать обробка переривань, керування пам'яттю, керування введенням-виведенням. До надійності та продуктивності ядра висувають підвищені вимоги.

Основною характерною ознакою ядра є те, що воно виконується у привілейованому режимі. Розглянемо особливості цього режиму.

Для забезпечення ефективного керування ресурсами комп'ютера ОС повинна мати певні привілеї щодо прикладних програм. Треба, щоб прикладні програми не втручалися в роботу ОС, і водночас ОС повинна мати можливість втрутитися в роботу будь-якої програми, наприклад для перемикання процесора або розв'язання конфлікту в боротьбі за ресурси. Для реалізації таких привілеїв потрібна апаратна підтримка: процесор має підтримувати принаймні два режими роботи — *привілейований* (захищений режим, режим ядра, kernel mode) і *режим користувача* (user mode). У режимі користувача недопустимі команди, які є критичними для роботи системи (перемикання задач, звертання до пам'яті за заданими межами, доступ до пристроїв введення-виведення тощо).

Розглянемо, яким чином використовуються різні режими процесора під час взаємодії між ядром і застосуваннями.

Після завантаження ядро перемикає процесор у привілейований режим і отримує цілковитий контроль над комп'ютером. Кожне застосування запускається і виконується в режимі користувача, де воно не має доступу до ресурсів ядра й інших програм. Коли потрібно виконати дію, реалізовану в ядрі, застосування робить *системний виклик* (system call). Ядро перехоплює його, перемикає процесор у привілейований режим, виконує дію, перемикає процесор назад у режим користувача і повертає результат застосування.

Системний виклик виконується повільніше за виклик функції, реалізованої в режимі користувача, через те що процесор двічі перемикається між режимами. Для підвищення продуктивності в деяких ОС частина функціональності реалізована в режимі користувача, тому для доступу до неї системні виклики використовувати не потрібно.

2.1.3. Системне програмне забезпечення

Окрім ядра, важливими складниками роботи ОС є також застосування режиму користувача, які виконують системні функції. До такого *системного програмного забезпечення* належать:

- системні програми (утиліти), наприклад: командний інтерпретатор, програми резервного копіювання та відновлення даних, засоби діагностики й адміністрування;
- системні бібліотеки, у яких реалізовані функції, що використовуються у застосуваннях користувача.

Системне програмне забезпечення може розроблятися й постачатися окремо від ОС. Наприклад, може бути кілька реалізацій командного інтерпретатора, засобів резервного копіювання тощо. Системні програми і бібліотеки взаємодіють з ядром у такий самий спосіб, як і прикладні програми.

2.2. Реалізація архітектури операційних систем

У цьому розділі розглянуто кілька підходів до реалізації архітектури операційних систем. У реальних ОС звичайно використовують деяку комбінацію цих підходів.

2.2.1. Монолітні системи

ОС, у яких усі базові функції сконцентровані в ядрі, називають *монолітними системами*. У разі реалізації монолітного ядра ОС стає продуктивнішою (процесор не перемикається між режимами під час взаємодії між її компонентами), але менш надійною (весь її код виконується у привілейованому режимі, і помилка в кожному з компонентів є критичною).

Монолітність ядра не означає, що всі його компоненти мають постійно перебувати у пам'яті. Сучасні ОС дають можливість динамічно розміщувати в адресному просторі ядра фрагменти коду (*модулі ядра*). Реалізація модулів ядра дає можливість також досягти його розширюваності (для додання нової функціональності досить розробити і завантажити у пам'ять відповідний модуль).

2.2.2. Багаторівневі системи

Компоненти *багаторівневих ОС* утворюють ієрархію рівнів (шарів, layers), кожен з яких спирається на функції попереднього рівня. Найнижчий рівень безпосередньо взаємодіє з апаратним забезпеченням, на найвищому рівні реалізуються системні виклики.

У традиційних багаторівневих ОС передача керування з верхнього рівня на нижній реалізується як системний виклик. Верхній рівень повинен мати права на виконання цього виклику, перевірка цих прав виконується за підтримки апаратного забезпечення. Прикладом такої системи є ОС Multics, розроблена в 60-ті роки. Практичне застосування цього підходу сьогодні обмежене через низьку продуктивність.

Рівні можуть виділятися й у монолітному ядрі; у такому разі вони підтримуються програмно і спричиняють спрощення реалізації системи. У монолітному ядрі визначають рівні, перелічені нижче.

- *Засоби абстрагування від устаткування*, які взаємодіють із апаратним забезпеченням безпосередньо, звільняючи від реалізації такої взаємодії інші компоненти системи.
- *Базові засоби ядра*, які відповідають за найфундаментальніші, найпростіші дії ядра, такі як запис блоку даних на диск. За допомогою цих засобів виконуються вказівки верхніх рівнів, пов'язані з керуванням ресурсами.
- *Засоби керування ресурсами* (або менеджери ресурсів), що реалізують основні функції ОС (керування процесами, пам'яттю, введенням-виведенням тощо). На цьому рівні приймаються найважливіші рішення з керування ресурсами, які виконуються з використанням базових засобів ядра.
- *Інтерфейс системних викликів*, який служить для реалізації зв'язку із системним і прикладним програмним забезпеченням.

Розмежування базових засобів ядра і менеджерів ресурсів відповідає відокремленню механізмів від політики в архітектурі системи. Базові засоби ядра визначають механізми функціонування системи, менеджери ресурсів реалізують політику.

2.2.3. Системи з мікроядром

Один із напрямів розвитку сучасних ОС полягає в тому, що у привілейованому режимі реалізована невелика частка функцій ядра, які є *мікроядром* (microkernel). Інші функції ОС виконуються процесами режиму користувача (серверними процесами, серверами). Сервери можуть відповідати за підтримку файлової системи, за роботу із процесами, пам'яттю тощо.

Мікроядро здійснює зв'язок між компонентами системи і виконує базовий розподіл ресурсів. Щоб виконати системний виклик, процес (клієнтський процес, клієнт) звертається до мікроядра. Мікроядро посилає серверу запит, сервер виконує роботу і пересилає відповідь назад, а мікроядро переправляє його клієнтові (рис. 2.1). Клієнтами можуть бути не лише процеси користувача, а й інші модулі ОС.

Перевагами мікроядрового підходу є:

- невеликі розміри мікроядра, що спрощує його розробку й налагодження;
- висока надійність системи, внаслідок того що сервери працюють у режимі користувача й у них немає прямого доступу до апаратного забезпечення;
- більша гнучкість і розширюваність системи (непотрібні компоненти не займають місця в пам'яті, розширення функціональності системи зводиться до додавання в неї нового сервера);
- можливість адаптації до умов мережі (спосіб обміну даними між клієнтом і сервером не залежить від того, зв'язані вони мережею чи перебувають на одному комп'ютері).

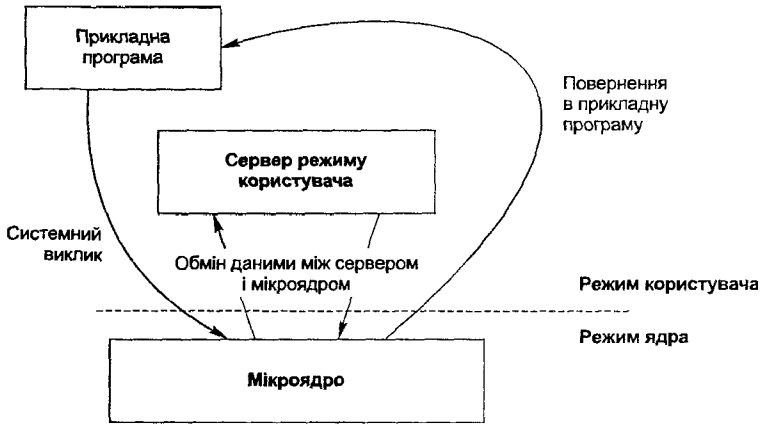

Рис. 2.1. Виконання системного виклику в архітектурі з мікроядром

Головним недоліком мікроядрового підходу є зниження продуктивності. Замість двох перемикань режиму процесора у разі системного виклику відбувається чотири (два — під час обміну між клієнтом і мікроядром, два — між сервером та мікроядром).

Зазначений недолік є, швидше, теоретичним, на практиці продуктивність і надійність мікроядра залежать насамперед від якості його реалізації. Так, в ОС QNX мікроядро займає кілька кілобайтів пам'яті й забезпечує мінімальний набір функцій, при цьому система за продуктивністю відповідає ОС реального часу.

2.2.4. Концепція віртуальних машин

У *системах віртуальних машин* програмним шляхом створюють копії апаратного забезпечення (відбувається його емуляція). Ці копії (*віртуальні машини*) працюють паралельно, на кожній із них функціонує програмне забезпечення, з яким взаємодіють прикладні програми і користувачі.

Уперше концепція віртуальних машин була реалізована в 70-ті роки в операційній системі VM фірми IBM. У СРСР варіант цієї системи (VM/370) був широко розповсюджений у 80-ті роки і мав назву Система віртуальних машин ЄС ЕОМ (СВМ ЄС). Розглянемо архітектуру цієї ОС [8, 44], що показана на рис. 2.2.

Ядро системи, яке називалося монітором віртуальних машин (VM Monitor, МВМ), виконувалося на фізичній машині, безпосередньо взаємодіючи з її апаратним забезпеченням. Монітор реалізовував набір віртуальних машин (ВМ). Кожна ВМ була точною копією апаратного забезпечення, на ній могла бути запущена будь-яка ОС, розроблена для цієї архітектури. Найчастіше на ВМ встановлювали спеціальну однокористувацьку ОС CMS (підсистема діалогової обробки, ПДО). На різних ВМ могли одночасно функціонувати різні ОС.

Коли програма, написана для ПДО, виконувала системний виклик, його перехоплювала копія ПДО, запущена на відповідній віртуальній машині. Потім ПДО виконувала відповідні апаратні інструкції, наприклад інструкції введення-виведення для читання диска. Ці інструкції перехоплював МВМ і перетворював їх на апаратні інструкції фізичної машини.

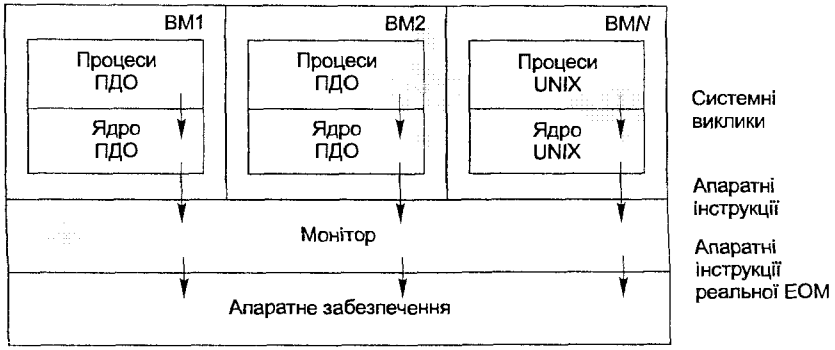

Рис. 2.2. Архітектура VM/370 [44]

Віртуальні машини спільно використовували ресурси реального комп'ютера; наприклад, дисковий простір розподілявся між ними у вигляді віртуальних дисків, названих мінідисками. ОС, запущена у ВМ, використовувала мінідиски так само, як фізичні диски.

Сьогодні концепція віртуальних машин застосовується і в прикладному програмному забезпеченні; опис відповідних рішень (програмних емуляторів апаратного забезпечення, технології керованого коду) можна знайти на сайті супроводу.

2.3. Операційна система та її оточення

Із означення ОС випливає, що вона реалізує зв'язок між апаратним забезпеченням комп'ютера (через інтерфейс апаратного забезпечення) і програмами користувача (через інтерфейс прикладного програмування). У цьому розділі розглянуто особливості реалізації та використання цих інтерфейсів.

2.3.1. Взаємодія ОС і апаратного забезпечення

Взаємодію ОС і апаратного забезпечення слід розглядати з двох боків. З одного боку, ОС повинна реалізовувати засоби взаємодії з апаратним забезпеченням, з іншого – архітектуру комп'ютера треба проектувати з урахуванням того, що на комп'ютері функціонуватиме ОС.

Інтерфейс апаратного забезпечення має бути повністю схований від прикладних програм і користувачів, усю роботу з ним виконує ОС. Розглянемо особливості апаратної підтримки цього інтерфейсу і його використання в сучасних ОС.

Засоби апаратної підтримки операційних систем

Сучасні апаратні архітектури комп'ютерів реалізують базові засоби підтримки операційних систем. До них належать: система переривань, привілейований режим процесора, засоби перемикання задач, підтримка керування пам'яттю (механізми трансляції адрес, захист пам'яті), системний таймер, захист пристроїв введення-виведення, базова система введення-виведення (BIOS). Розглянемо ці засоби докладніше.

Система переривань є основним механізмом, що забезпечує функціонування ОС. За допомогою переривань процесор отримує інформацію про події, не пов'язані з основним циклом його роботи (отриманням інструкцій з пам'яті та їхнім виконанням). Переривання бувають двох типів: *апаратні* і *програмні*.

Апаратне переривання – це спеціальний сигнал (запит переривання, IRQ), що передається процесору від апаратного пристрою.

До апаратних переривань належать:

- переривання введення-виведення, що надходять від контролера периферійного пристрою; наприклад, таке переривання генерує контролер клавіатури при натисканні на клавішу;
- переривання, пов'язані з апаратними або програмними помилками (такі переривання виникають, наприклад, у разі збою контролера диска, доступу до забороненої області пам'яті або ділення на нуль).

Програмні переривання генерує прикладна програма, виконуючи спеціальну *інструкцію переривання*. Така інструкція є в системі команд більшості процесорів. Обробка програмних переривань процесором не відрізняється від обробки апаратних переривань.

Якщо переривання відбулося, то процесор негайно передає керування спеціальній процедурі – *оброблювачеві переривання*. Після виходу з оброблювача процесор продовжує виконання інструкцій перерваної програми. Розрізняють два типи переривань залежно від того, яка інструкція буде виконана після виходу з оброблювача: для *відмов* (faults) повторюється інструкція, що спричинила переривання, для *пасток* (traps) – виконується наступна інструкція. Усі переривання введення-виведення і програмні переривання належать до категорії пасток, більшість переривань через помилки є відмовами.

За встановлення оброблювачів переривань зазвичай відповідає ОС. Можна сказати, що сучасні ОС керовані перериваннями (interrupt-driven), бо, якщо ОС не зайнята виконанням якої-небудь задачі, вона очікує на переривання, яке й залучає її до роботи.

Для реалізації *привілейованого режиму процесора* в одному з його регістрів передбачено спеціальний біт (біт режиму), котрий показує, у якому режимі перебуває процесор. У разі програмного або апаратного переривання процесор автоматично перемикається у привілейований режим, і саме тому ядро ОС (яке складається з оброблювачів переривань) завжди отримує керування в цьому режимі. За будь-якої спроби безпосередньо виконати привілейовану інструкцію в режимі користувача відбувається апаратне переривання.

Засоби перемикання задач дають змогу зберігати вміст регістрів процесора (контекст задачі) у разі припинення задачі та відновлювати дані перед її подальшим виконанням.

Механізм трансляції адрес забезпечує перетворення адрес пам'яті, з якими працює програма, в адреси фізичної пам'яті комп'ютера. Апаратне забезпечення генерує фізичну адресу, використовуючи спеціальні таблиці трансляції.

Захист пам'яті забезпечує перевірку прав доступу до пам'яті під час кожної спроби його отримати. Засоби захисту пам'яті інтегровані з механізмами трансляції адрес: у таблицях трансляції утримується інформація про права, необхідні для їхнього використання, і про *ліміт* (розміри ділянки пам'яті, до якої можна

отримати доступ з їхньою допомогою). Неможливо одержати доступ до пам'яті поверх ліміту або за відсутності прав на використання таблиці трансляції.

Системний таймер є апаратним пристроєм, який генерує *переривання таймера* через певні проміжки часу. Такі переривання обробляє ОС; інформацію від таймера найчастіше використовують для визначення часу перемикання задач.

Захист пристроїв введення-виведення ґрунтується на тому, що всі інструкції введення-виведення визначені як привілейовані. Прикладні програми здійснюють введення-виведення не прямо, а за посередництвом ОС.

Базова система введення-виведення (BIOS) – службовий програмний код, що зберігається в постійному запам'ятовувальному пристрої і призначений для ізоляції ОС від конкретного апаратного забезпечення. Зазначимо, що засоби BIOS не завжди дають змогу використати всі можливості архітектури: наприклад, процедури BIOS для архітектури IA-32 не працюють у захищеному режимі. Тому сучасні ОС використовують їх тільки для початкового завантаження системи.

Апаратна незалежність і здатність до перенесення ОС

Як було згадано в розділі 2.2.2, компоненти ядра, які відповідають за безпосередній доступ до апаратного забезпечення, виділено в окремий рівень абстрагування від устаткування, що взаємодіє з іншою частиною системи через стандартні інтерфейси. Тим самим спрощується досягнення апаратної незалежності ОС.

Рівень абстрагування від устаткування відображає такі особливості архітектури, як число процесорів, типи їхніх регістрів, розрядність і організація пам'яті тощо. Що більше відмінностей між апаратними архітектурами, для яких призначена ОС, то складніша розробка коду цього рівня.

Крім рівня абстрагування від устаткування, від апаратного забезпечення залежать драйвери зовнішніх пристроїв. Такі драйвери проектують заздалегідь як апаратно-залежні, їх можна додавати та вилучати за потребою; для доступу до них зазвичай використовують універсальний інтерфейс.

Здатність до перенесення ОС визначається обсягом робіт, необхідних для того, щоб система могла працювати на новій апаратній платформі. ОС з такими властивостями має відповідати певним вимогам.

- Більша частина коду операційної системи має бути написана мовою високого рівня (звичайно для цього використовують мови C і C++, компілятори яких розроблені для більшості архітектур). Використання мови асемблера допустиме лише тоді, коли продуктивність компонента є критичною для системи.
- Код, що залежить від апаратного забезпечення (рівень абстрагування від устаткування) має бути відокремлений від іншої частини системи так, щоб у разі переходу на іншу архітектуру потрібно було переписувати тільки цей рівень.

2.3.2. Взаємодія ОС і виконуваного програмного коду

У роботі в режимі користувача часто необхідне виконання дій, реалізованих у ядрі ОС (наприклад, під час запису на диск із прикладної програми). Для цього треба забезпечити взаємодію коду режиму користувача та ОС. Розглянемо особливості такої взаємодії.

Системні виклики та інтерфейс програмування застосувань

Системний виклик – це засіб доступу до певної функції ядра операційної системи із прикладних програм. Набір системних викликів визначає дії, які ядро може виконати за запитом процесів користувача. Системні виклики задають інтерфейс між прикладною програмою і ядром ОС.

Розглянемо послідовність виконання системного виклику.

1. Припустимо, що для процесу, який виконується в режимі користувача, потрібна функція, реалізована в ядрі системи.
2. Для того щоб звернутися до цієї функції, процес має передати керування ядру ОС, для чого необхідно задати параметри виклику і виконати програмне переривання (*інструкцію системного виклику*). Відбувається перехід у привілейований режим.
3. Після отримання керування ядро зчитує параметри виклику і визначає, що потрібно зробити.
4. Після цього ядро виконує потрібні дії, зберігає в пам'яті значення, які слід повернути, і передає керування програмі, що його викликала. Відбувається перехід назад у режим користувача.
5. Програма зчитує з пам'яті повернені значення і продовжує свою роботу.

Як бачимо, кожний системний виклик спричиняє перехід у привілейований режим і назад (у мікроядровій архітектурі, як було зазначено вище, таких переходів може бути і більше).

Розглянемо способи передачі параметрів у системний виклик. До них належать:

- передача параметрів у регістрах процесора;
- занесення параметрів у певну ділянку пам'яті й передача покажчика на неї в регістрі процесора.

Системні виклики призначені для безпосереднього доступу до служб ядра ОС. На практиці вони не вичерпують (а іноді й не визначають) ті функції, які можна використати у прикладних програмах для доступу до служб ОС або засобів системних бібліотек. Для позначення цього набору функцій використовують термін *інтерфейс програмування застосувань* (Application Programming Interface, API).

Взаємозв'язок між функціями API і системними викликами неоднаковий у різних ОС.

По-перше, кожному системному виклику може бути поставлена у відповідність бібліотечна функція, єдиним завданням якої є виконання цього виклику. Таку функцію називають *пакувальником системного виклику* (system call wtapper). Для програміста в цьому разі набір функцій API виглядає як сукупність таких пакувальників і додаткових функцій, реалізованих бібліотеками повністю або частково в режимі користувача. Це рішення прийняте за основу в UNIX; у такому разі прийнято говорити про використання системних викликів у прикладних програмах (насправді у програмах викликають пакувальники системних викликів).

По-друге, можна надати для використання у прикладних програмах універсальний інтерфейс програмування застосувань (API режиму користувача) і повністю сховати за ним набір системних викликів. Для програміста кожна функція такого API є бібліотечною функцією режиму користувача, пакувальника в цьому

разі немає, відомості про системні виклики є деталями реалізації ОС. Це властиве Windows-системам, де подібний універсальний набір функцій називають *Win32 API* [31, 50].

Програмна сумісність

Дотепер ми розглядали виконання в ОС програм, розроблених спеціально для неї. Іноді буває необхідно виконати в середовищі ОС програми, розроблені для інших ОС і, можливо, для іншої апаратної архітектури. У цьому разі виникає проблема *програмної сумісності*.

Програмна сумісність означає можливість виконувати в середовищі однієї операційної системи програми, розроблені для іншої ОС. Розрізняють *сумісність на рівні вихідних текстів* (можливість перенесення вихідних текстів) та *бінарну сумісність* (можливість перенесення виконуваного коду).

Для сумісності на рівні вихідних текстів необхідно, щоб для всіх ОС існувала реалізація компілятора мови і API, що його використовує програма.

Сьогодні таку сумісність забезпечує стандартизація розробки програмного забезпечення, а саме:

- наявність стандарту на мови програмування (насамперед на C і C++) і стандартних компіляторів;
- наявність стандарту на інтерфейс операційної системи (API).

Робота зі стандартизації інтерфейсу операційних систем відбувається у рамках проекту POSIX (Portable Operating System Interface). Найбільш важливим стандартом є POSIX 1003.1 [3], який описує набір бібліотечних процедур (таких, як відкриття файла, створення нового процесу тощо), котрі мають бути реалізовані в системі. Цей процес стандартизації триває дотепер, останньою редакцією стандарту є базова специфікація Open Group/IEEE [103]. Зазначені стандарти відображають традиційний набір засобів, реалізованих в UNIX-сумісних системах.

Завдання забезпечення бінарної сумісності виникає тоді, коли потрібно запустити на виконання файл прикладної програми у середовищі іншої операційної системи. Таке завдання значно складніше в реалізації, найпоширеніший підхід до його розв'язання – *емуляція середовища виконання*. У цьому разі програма запускається під керуванням іншої програми – *емулятора*, який забезпечує динамічне перетворення інструкцій програми в інструкції потрібної архітектури. Прикладом такого емулятора є програма wine, яка дає можливість запускати програми, розроблені для Win32 API, у середовищі Linux через емуляцію функцій Win32 API системними викликами Linux.

2.4. Особливості архітектури: UNIX і Linux

2.4.1. Базова архітектура UNIX

UNIX є прикладом досить простої архітектури ОС. Більша частина функціональності цієї системи міститься в ядрі, ядро спілкується із прикладними програмами за допомогою системних викликів. Базова структура класичного ядра UNIX зображена на рис. 2.3 (див., наприклад, [33, 59]).

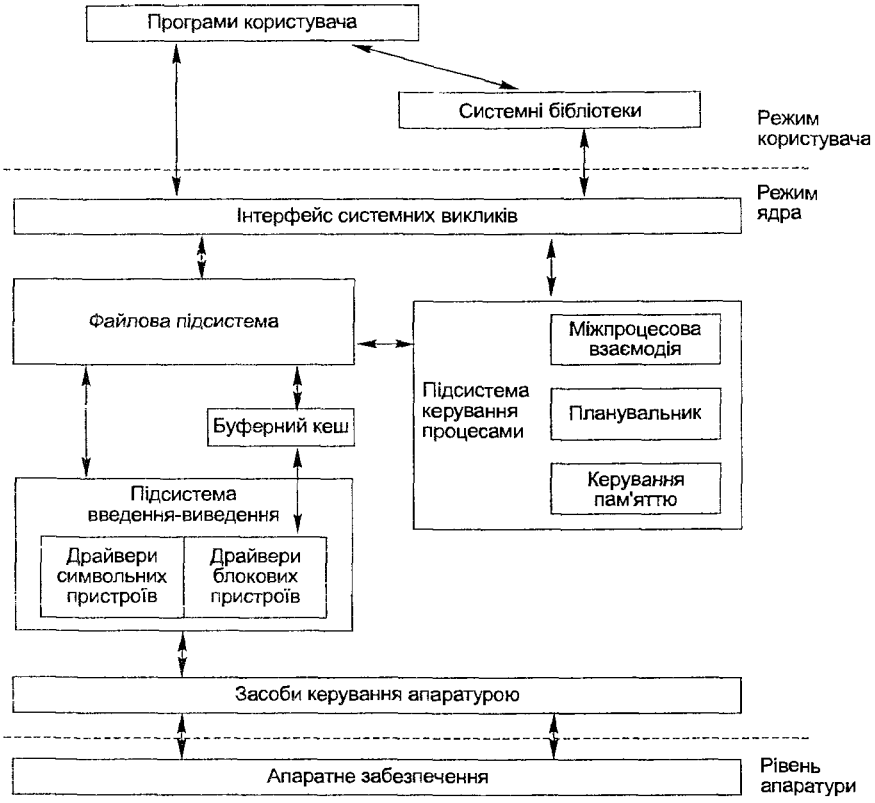

Рис. 2.3. Архітектура UNIX

Система складається із трьох основних компонентів: підсистеми керування процесами, файлової підсистеми та підсистеми введення-виведення.

Підсистема керування процесами контролює створення та вилучення процесів, розподілення системних ресурсів між ними, міжпроцесову взаємодію, керування пам'яттю.

Файлова підсистема забезпечує єдиний інтерфейс доступу до даних, розташованих на дискових накопичувачах, і до периферійних пристроїв. Такий інтерфейс є однією з найважливіших особливостей UNIX. Одні й ті самі системні виклики використовують як для обміну даними із диском, так і для виведення на термінал або принтер (програма працює із принтером так само, як із файлом). При цьому файлова система переадресовує запити відповідним модулям підсистеми введення-виведення, а ті – безпосередньо периферійним пристроям. Крім того, файлова підсистема контролює права доступу до файлів, які значною мірою визначають привілеї користувача в системі.

Підсистема введення-виведення виконує запити файлової підсистеми, взаємодіючи з драйверами пристроїв. В UNIX розрізняють два типи пристроїв: символьні (наприклад, принтер) і блокові (наприклад, жорсткий диск). Основна відмінність між ними полягає в тому, що блоковий пристрій допускає прямий

доступ. Для підвищення продуктивності роботи із блоковими пристроями використовують буферний кеш — ділянку пам'яті, у якій зберігаються дані, зчитані з диска останніми. Під час наступних звертань до цих даних вони можуть бути отримані з кеша.

Сучасні UNIX-системи дещо відрізняються за своєю архітектурою.

- У них виділено окремий менеджер пам'яті, відповідальний за підтримку віртуальної пам'яті.
- Стандартом для реалізації інтерфейсу файлової системи є *віртуальна файлова система*, що абстрагує цей інтерфейс і дає змогу організувати підтримку різних типів файлових систем.
- У цих системах підтримується багатопроцесорна обробка, а також багатопотоковість.

Базові архітектурні рішення, такі як доступ до всіх пристроїв введення-виведення через інтерфейс файлової системи або організація системних викликів, залишаються незмінними в усіх реалізаціях UNIX.

2.4.2. Архітектура Linux

В ОС Linux можна виділити три основні частини:

- *ядро*, яке реалізує основні функції ОС (керування процесами, пам'яттю, введенням-виведенням тощо);
- *системні бібліотеки*, що визначають стандартний набір функцій для використання у застосуваннях (виконання таких функцій не потребує переходу в привілейований режим);
- *системні утиліти* (прикладні програми, які виконують спеціалізовані задачі).

Призначення ядра Linux і його особливості

Linux реалізує технологію монолітного ядра. Весь код і структури даних ядра перебувають в одному адресному просторі. У ядрі можна виділити кілька функціональних компонентів [63].

- *Планувальник процесів* — відповідає за реалізацію багатозадачності в системі (обробка переривань, робота з таймером, створення і завершення процесів, перемикання контексту).
- *Менеджер пам'яті* — виділяє окремий адресний простір для кожного процесу і реалізує підтримку віртуальної пам'яті.
- *Віртуальна файлова система* — надає універсальний інтерфейс взаємодії з різними файловими системами та пристроями введення-виведення.
- *Драйвери пристроїв* — забезпечують безпосередню роботу з периферійними пристроями. Доступ до них здійснюється через інтерфейс віртуальної файлової системи.
- *Мережний інтерфейс* — забезпечує доступ до реалізації мережних протоколів і драйверів мережних пристроїв.
- *Підсистема міжпроцесової взаємодії* — пропонує механізми, які дають змогу різним процесам у системі обмінюватися даними між собою.

Деякі із цих підсистем є логічними компонентами системи, вони завантажуються у пам'ять разом із ядром і залишаються там постійно. Компоненти інших

підсистем (наприклад, драйвери пристроїв) вигідно реалізовувати так, щоб їхній код міг завантажуватися у пам'ять на вимогу. Для розв'язання цього завдання Linux підтримує концепцію модулів ядра.

Модулі ядра

Ядро Linux дає можливість на вимогу завантажувати у пам'ять і вивантажувати з неї окремі секції коду. Такі секції називають *модулями ядра* (kernel modules) [30] і виконують у привілейованому режимі.

Модулі ядра надають низку переваг.

- Код модулів може завантажуватися в пам'ять у процесі роботи системи, що спрощує налагодження компонентів ядра, насамперед драйверів.
- З'являється можливість змінювати набір компонентів ядра під час виконання: ті з них, які в цей момент не використовуються, можна не завантажувати у пам'ять.
- Модулі є винятком із правила, за яким код, що розширює функції ядра, відповідно до ліцензії Linux має бути відкритим. Це дає змогу виробникам апаратного забезпечення розробляти драйвери під Linux, навіть якщо не заплановано надавати доступ до їхнього вихідного коду.

Підтримка модулів у Linux складається із трьох компонентів.

- Засоби керування модулями дають можливість завантажувати модулі у пам'ять і здійснювати обмін даними між модулями та іншою частиною ядра.
- Засоби реєстрації драйверів дозволяють модулям повідомляти іншу частину ядра про те, що новий драйвер став доступним.
- Засоби розв'язання конфліктів дають змогу драйверам пристроїв резервувати апаратні ресурси і захищати їх від випадкового використання іншими драйверами.

Один модуль може зареєструвати кілька драйверів, якщо це потрібно (наприклад, для двох різних механізмів доступу до пристрою).

Модулі можуть бути завантажені заздалегідь — під час старту системи (завантажувальні модулі) або у процесі виконання програми, яка викликає їхні функції. Після завантаження код модуля перебуває в тому ж самому адресному просторі, що й інший код ядра. Помилка в модулі є критичною для системи.

Особливості системних бібліотек

Системні бібліотеки Linux є *динамічними бібліотеками*, котрі завантажуються у пам'ять тільки тоді, коли у них виникає потреба. Вони виконують ряд функцій:

- реалізацію пакувальників системних викликів;
- розширення функціональності системних викликів (до таких бібліотек належить бібліотека введення-виведення мови C, яка реалізує на основі системних викликів такі функції, як `printf()`);
- реалізацію службових функцій режиму користувача (сортування, функції обробки рядків тощо).

Застосування користувача

Застосування користувача в Linux використовують функції із системних бібліотек і через них взаємодіють із ядром за допомогою системних викликів.

2.5. Особливості архітектури: Windows XP

У цьому розділі ми розглянемо основні компоненти Windows XP [14, 44], які зображені на рис. 2.4.

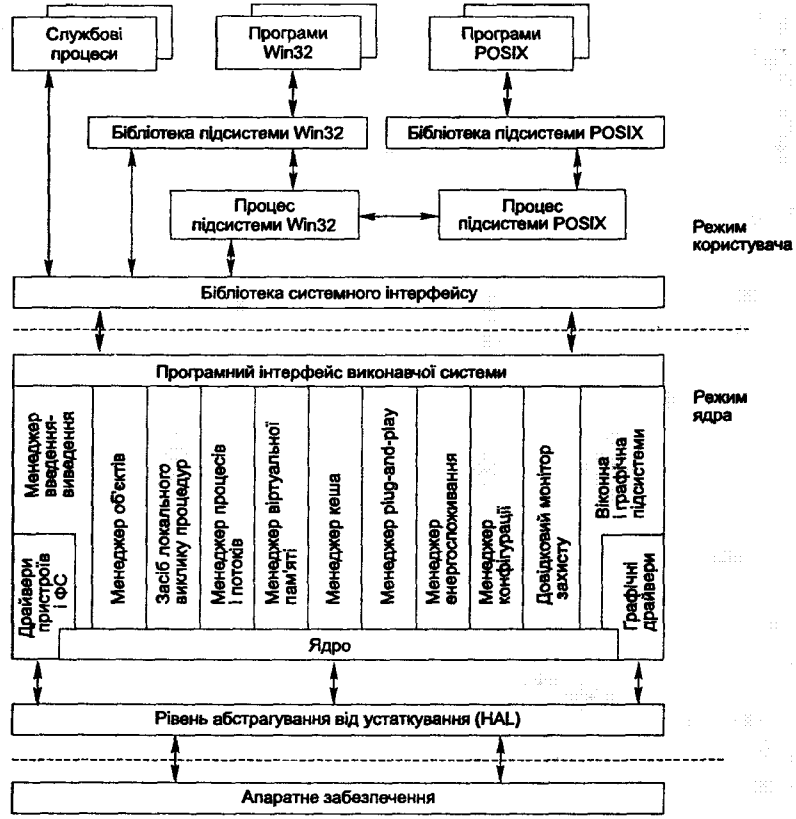

Рис. 2.4. Базові компоненти Windows XP

Деякі компоненти Windows XP виконуються у привілейованому режимі, інші компоненти – у режимі користувача. Ми почнемо розгляд системи з компонентів режиму ядра.

2.5.1. Компоненти режиму ядра

У традиційному розумінні (див. розділ 2.1.2) ядро ОС містить усі компоненти привілейованого режиму, однак у Windows XP поняття ядра закріплене тільки за одним із цих компонентів.

Рівень абстрагування від устаткування

У Windows XP реалізовано рівень абстрагування від устаткування (у цій системі його називають HAL, hardware abstraction layer). Для різних апаратних конфігурацій фірма Microsoft або сторонні розробники можуть постачати різні реалізації HAL.

Хоча код HAL є дуже ефективним, його використання може знижувати продуктивність застосувань мультимедіа. У такому разі використовують спеціальний пакет DirectX, який дає змогу прикладним програмам звертатися безпосередньо до апаратного забезпечення, обминаючи HAL та інші рівні системи.

Ядро

Ядро Windows XP відповідає за базові операції системи. До його основних функцій належать:

- перемикання контексту, збереження і відновлення стану потоків;
- планування виконання потоків;
- реалізація засобів підтримки апаратного забезпечення, складніших за засоби HAL (наприклад, передача керування обробловачам переривань).

Ядро Windows XP відповідає базовим службам ОС і надає набір механізмів для реалізації політики керування ресурсами.

Основним завданням ядра є якомога ефективніше завантаження процесорів системи. Ядро постійно перебуває в пам'яті, послідовність виконання його інструкцій може порушити тільки переривання (під час виконання коду ядра багатозадачність не підтримується). Для прискорення роботи ядро ніколи не перевіряє правильність параметрів, переданих під час виклику його функцій.

Windows XP не можна віднести до якогось певного класу ОС. Наприклад, хоча за функціональністю ядро системи відповідає поняттю мікроядра, для самої ОС не характерна класична мікроядрова архітектура, оскільки у привілейованому режимі виконуються й інші її компоненти.

Виконавча система

Виконавча система (ВС) Windows XP (Windows XP Executive) – це набір компонентів, відповідальних за найважливіші служби ОС (керування пам'яттю, процесами і потоками, введенням-виведенням тощо).

Компонентами ВС є передусім базові засоби підтримки. Ці засоби використовують у всій системі.

- *Менеджер об'єктів* – відповідає за розподіл ресурсів у системі, підтримуючи їхнє універсальне подання через об'єкти.
- *Засіб локального виклику процедур* (LPC) – забезпечує механізм зв'язку між процесами і підсистемами на одному комп'ютері.

Інші компоненти ВС реалізують найважливіші служби Windows XP. Зупинимося на деяких із них.

- *Менеджер процесів і потоків* – створює та завершує процеси і потоки, а також розподіляє для них ресурси.
- *Менеджер віртуальної пам'яті* – реалізує керування пам'яттю в системі, насамперед підтримку віртуальної пам'яті.
- *Менеджер введення-виведення* – керує периферійними пристроями, надаючи іншим компонентам апаратно-незалежні засоби введення-виведення. Цей менеджер реалізує єдиний інтерфейс для драйверів пристроїв.

- *Менеджер кеша* – керує кешуванням для системи введення-виведення. Часто використовувані блоки диска тимчасово зберігаються в пам'яті, наступні операції введення-виведення звертаються до цієї пам'яті, внаслідок чого підвищується продуктивність.
- *Менеджер конфігурації* – відповідає за підтримку роботи із *системним реєстром* (registry) – ієрархічно організованим сховищем інформації про налаштування системи і прикладних програм.
- *Довідковий монітор захисту* – забезпечує політику безпеки на ізольованому комп'ютері, тобто захищає системні ресурси.

Драйвери пристроїв

У Windows XP драйвери не обов'язково пов'язані з апаратними пристроями. Застосування, якому потрібні засоби, доступні в режимі ядра, завжди варто оформляти як драйвер. Це пов'язане з тим, що для зовнішніх розробників режим ядра доступний тільки з коду драйверів. Докладніше реалізацію драйверів Windows XP буде розглянуто в розділі 15.

Віконна і графічна підсистеми

Віконна і графічна підсистеми відповідають за інтерфейс користувача – роботу з вікнами, елементами керування і графічним виведенням.

- *Менеджер вікон* – реалізує високорівневі функції. Він керує віконним виведенням, обробляє введення з клавіатури або миші й передає застосуванням повідомлення користувача.
- *Інтерфейс графічних пристроїв* (Graphical Device Interface, GDI) – складається з набору базових операцій графічного виведення, які не залежать від конкретного пристрою (креслення ліній, відображення тексту тощо).
- *Драйвери графічних пристроїв* (відеокарт, принтерів тощо) – відповідають за взаємодію з контролерами цих пристроїв.

Під час створення вікон або елементів керування запит надходить до менеджера вікон, який для виконання базових графічних операцій звертається до GDI. Потім запит передається драйверу пристрою, затим – апаратному забезпеченню через HAL.

2.5.2. Компоненти режиму користувача

Компоненти режиму користувача не мають прямого доступу до апаратного забезпечення, їхній код виконується в ізольованому адресному просторі. Більша частина коду режиму користувача перебуває в динамічних бібліотеках, які у Windows називають DLL (dynamic-link libraries).

Бібліотека системного інтерфейсу

Для доступу до засобів режиму ядра в режимі користувача необхідно звертатися до функцій бібліотеки системного інтерфейсу (ntdll.dll). Ця бібліотека надає набір функцій-перехідників, кожній з яких відповідає функція режиму ядра (системний виклик). Застосування зазвичай не викликають такі функції безпосередньо, за це відповідають підсистеми середовища.

Підсистеми середовища

Підсистеми середовища надають застосуванням користувача доступ до служб операційної системи, реалізуючи відповідний API. Ми зупинимося на двох підсистемах середовища: Win32 і POSIX.

Підсистема Win32, яка реалізує Win32 API, є обов'язковим компонентом Windows XP. До неї входять такі компоненти:

- процес підсистеми Win32 (csrss.exe), що відповідає, зокрема, за реалізацію текстового (консольного) введення-виведення, створення і знищення процесів та потоків;
- бібліотеки підсистеми Win32, які надають прикладним програмам функції Win32 API. Найчастіше використовують бібліотеки gdi32.dll (низькорівневі графічні функції, незалежні від пристрою), user32.dll (функції інтерфейсу користувача) і kernel32.dll (функції, реалізовані у ВС і ядрі).

Після того як застосування звернеться до функції Win32 API, спочатку буде викликана відповідна функція з бібліотеки підсистеми Win32. Розглянемо варіанти виконання такого виклику.

1. Якщо функції потрібні тільки ресурси її бібліотеки, виклик повністю виконується в адресному просторі застосування без переходу в режим ядра.
2. Якщо потрібен перехід у режим ядра, з коду бібліотеки підсистеми виконується системний виклик. Так відбувається у більшості випадків, наприклад під час створення вікон або елементів керування.
3. Функція бібліотеки підсистеми може звернутися до процесу підсистеми Win32, при цьому:
   - коли потрібна тільки функціональність, реалізована даним процесом, переходу в режим ядра не відбувається;
   - коли потрібна функціональність режиму ядра, процес підсистеми Win32 виконує системний виклик аналогічно до варіанта 2.

Зазначимо, що до виходу Windows NT 4.0 (1996 рік) віконна і графічна підсистеми працювали в режимі користувача як частина процесу підсистеми Win32 (тобто виклики базових графічних функцій Win32 API оброблялися відповідно до варіанта 3). Надалі для підвищення продуктивності реалізацію цих підсистем було перенесено в режим ядра [14].

Підсистема POSIX працює в режимі користувача й реалізує набір функцій, визначених стандартом POSIX 1003.1. Оскільки застосування, або прикладні програми (applications), написані для однієї підсистеми, не можуть використати функції *інших, у POSIX-програмах* не можна користуватися засобами Win32 API (зокрема, графічними та мережними функціями), що знижує важливість цієї підсистеми. Підсистема POSIX не є обов'язковим компонентом Windows XP.

Наперед визначені системні процеси

Ряд важливих процесів користувача система запускає автоматично до закінчення завантаження. Розглянемо деякі з них.

- Менеджер сесій (Session Manager, smss.exe) створюється в системі першим. Він запускає інші важливі процеси (процес підсистеми Win32, процес реєстрації в системі тощо), а також відповідає за їхнє повторне виконання під час аварійного завершення.

- Процес реєстрації в системі (winlogon.exe) відповідає за допуск користувача в систему. Він відображає діалогове вікно для введення пароля, після введення передає пароль у підсистему безпеки і в разі успішної його верифікації запускає засоби створення сесії користувача.
- Менеджер керування службами (Service Control Manager, services.exe) відповідає за автоматичне виконання певних застосувань під час завантаження системи. Застосування, які будуть виконані при цьому, називають *службами* (services). Такі служби, як журнал подій, планувальник задач, менеджер друкування, постачають разом із системою. Крім того, є багато служб сторонніх розробників; так зазвичай реалізовують серверні застосування (сервери баз даних, веб-сервери тощо).

Застосування користувача

Застосування користувача можуть бути створені для різних підсистем середовища. Такі застосування використовують тільки функції відповідного API. Виклики цих функцій перетворюються в системні виклики за допомогою динамічних бібліотек підсистем середовища.

2.5.3. Об'єктна архітектура Windows XP

Керування ресурсами у Windows XP реалізується із застосуванням концепції об'єктів. Об'єкти надають універсальний інтерфейс для доступу до системних ресурсів, для яких передбачено спільне використання, зокрема таких, як процеси, потоки, файли і розподілювана пам'ять. Концепція об'єктів забезпечує важливі переваги.

- Імена об'єктів організовані в єдиний простір імен, де їх легко знаходити.
- Доступ до всіх об'єктів здійснюється однаково. Після створення нового об'єкта або після отримання доступу до наявного менеджер об'єктів повертає у застосування *дескриптор об'єкта* (object handle).
- Забезпечено захист ресурсів. Кожну спробу доступу до об'єкта розглядає підсистема захисту – без неї доступ до об'єкта, а отже і до ресурсу, отримати неможливо.

Менеджер об'єктів відповідає за створення, підтримку та ліквідацію об'єктів, задає єдині правила для їхнього іменування, збереження й забезпечення захисту. Підсистеми середовища звертаються до менеджера об'єктів безпосередньо або через інші сервіси ВС. Наприклад, під час запуску застосування підсистема Win32 викликає менеджер процесів для створення нового процесу. В свою чергу менеджер процесів звертається до менеджера об'єктів для створення об'єкта, що представляє процес.

Об'єкти реалізовано як структури даних в адресному просторі ядра. При перезавантаженні системи вміст усіх об'єктів губиться.

Особливості отримання доступу до об'єктів із процесів режиму користувача буде розглянуто в розділі 3.

Структура заголовка об'єкта

Об'єкти складаються з двох частин: заголовка і тіла об'єкта. У заголовку міститься інформація, загальна для всіх об'єктів, у тілі – специфічна для об'єктів конкретного типу.

До атрибутів заголовка об'єкта належать:

- ім'я об'єкта і його місце у просторі імен;
- дескриптор захисту (визначає права, необхідні для використання об'єкта);
- витрата квоти (ціна відкриття дескриптора об'єкта, дає змогу регулювати кількість об'єктів, які дозволено створювати);
- список процесів, що дістали доступ до дескрипторів об'єкта.

Менеджер об'єктів здійснює керування об'єктами на підставі інформації з їхніх заголовків.

Об'єкти типу

Формат і вміст тіла об'єкта визначається його типом. Новий тип об'єктів може бути визначений будь-яким компонентом ВС. Існує визначений набір типів об'єктів, які створюються під час завантаження системи (такі об'єкти, наприклад, відповідають процесам, відкритим файлам, пристроям введення-виведення).

Частина характеристик об'єктів є загальними для всіх об'єктів цього типу. Для зберігання відомостей про такі характеристики використовують спеціальні *об'єкти типу* (type objects). У такому об'єкті, зокрема, зберігають:

- ім'я типу об'єкта («процес», «потік», «відкритий файл» тощо);
- режими доступу (залежать від типу об'єкта: наприклад, для файла такими режимами можуть бути «читання» і «запис»).

Об'єкти типу недоступні в режимі користувача.

Методи об'єктів

Коли компонент ВС створює новий тип об'єкта, він може зареєструвати у диспетчері об'єктів один або кілька *методів*. Після цього диспетчер об'єктів викликає ці методи на певних етапах життєвого циклу об'єкта. Наведемо деякі з методів об'єктів:

- `open` — викликається при відкритті дескриптора об'єкта;
- `close` — викликається при закритті дескриптора об'єкта;
- `delete` — викликається перед вилученням об'єкта з пам'яті.

Покажчики на код реалізації методів також зберігаються в об'єктах типу.

Простір імен об'єктів

Усі імена об'єктів у ВС розташовані в глобальному просторі імен, тому будь-який процес може відкрити дескриптор об'єкта, вказавши його ім'я. Простір імен об'єктів має ієрархічну структуру, подібно до файлової системи. Аналогом каталогу файлової системи в такому просторі імен є *каталог об'єктів*. Він містить імена об'єктів (зокрема й інших каталогів). Перелічимо деякі наперед визначені імена каталогів:

- `Device` — імена пристроїв введення-виведення;
- `Driver` — завантажені драйвери пристроїв;
- `ObjectTypes` — об'єкти типів.

Простір імен об'єктів, як і окремі об'єкти, не зберігається після перезавантаження системи.

Висновки

- Архітектура ОС визначає набір її компонентів, а також порядок їхньої взаємодії один з одним та із зовнішнім середовищем.
- Найважливішим для вивчення архітектури ОС є поняття ядра системи. Основною характеристикою ядра є те, що воно виконується у привілейованому режимі.
- Основними типами архітектури ОС є монолітна архітектура й архітектура на базі мікроядра. Монолітна архітектура вимагає, щоб головні функції системи були сконцентровані в ядрі, найважливішою її перевагою є продуктивність. У системах на базі мікроядра в привілейованому режимі виконуються тільки базові функції, основними перевагами таких систем є надійність і гнучкість.
- Операційна система безпосередньо взаємодіє з апаратним забезпеченням комп'ютера. Сучасні комп'ютерні архітектури пропонують багато засобів підтримки роботи операційних систем. Для зв'язку з апаратним забезпеченням в ОС виділяється рівень абстрагування від устаткування.
- Операційна система взаємодіє із прикладними програмами. Вона надає набір системних викликів для доступу до функцій, реалізованих у ядрі. Для прикладних програм системні виклики разом із засобами системних бібліотек доступні через інтерфейс програмування застосувань (API).

Контрольні запитання та завдання

1. Перелічіть причини, за якими ядро ОС має виконуватися в привілейованому режимі процесора.
2. Чи може процесор переходити у привілейований режим під час виконання програми користувача? Чи може така програма виконуватися виключно в привілейованому режимі?
3. У чому полягає головний недолік традиційної багаторівневої архітектури ОС? Чи має такий недолік архітектура з виділенням рівнів у монолітному ядрі?
4. Чому перехід до використання мікроядрової архітектури може спричинити зниження продуктивності ОС?
5. Автор Linux Лінус Торвальдс стверджує, що мобільність Linux має поширюватися на системи з «прийнятною» (reasonable) архітектурою. Які апаратні засоби повинна підтримувати така архітектура?
6. Наведіть переваги і недоліки реалізації взаємодії прикладної програми з операційною системою в Linux і Windows XP.
7. Чи не суперечить використання модулів ядра принципам монолітної архітектури Linux? Поясніть свою відповідь.
8. Перелічіть переваги і недоліки архітектури ОС, відповідно до якої віконна і графічна підсистеми в Windows XP виконуються в режимі ядра.
9. Чому деякі діагностичні утиліти Windows XP складаються з прикладної програми і драйвера пристрою?

Розділ 3

Керування процесами і потоками

- Означення процесу та потоку
- Реалізація та використання моделі процесів і багатопотоковості
- Подання процесів і потоків в операційній системі
- Створення та завершення процесів і потоків
- Керування процесами та потоками в UNIX
- Керування процесами та потоками у Windows XP

У цьому розділі розглянемо дві основні абстракції операційної системи, які описують виконання програмного коду – процеси і потоки. Буде вивчено особливості їхньої реалізації в операційній системі, стани, в яких вони можуть перебувати, базові механізми роботи з ними (засоби створення, завершення, припинення виконання тощо).

У викладі орієнтуватимемося на сучасні ОС, для яких потоки є фундаментальними абстракціями системи нарівні з процесами.

3.1. Базові поняття процесів і потоків

3.1.1. Процеси і потоки в сучасних ОС

У сучасній операційній системі одночасно виконуються код ядра (що належить до його різних підсистем) і код програм користувача. При цьому відбуваються різні дії: одні програми і підсистеми виконують інструкції процесора, інші зайняті введенням-виведенням, ще деякі очікують на запити від користувача або інших застосувань. Для спрощення керування цими діями в системі доцільно виділити набір елементарних активних елементів і визначити інтерфейс взаємодії ОС із цими елементами. Коли активний елемент системи зв'язати із програмою, що виконується, ми прийдемо до поняття *процесу*.

Дамо попереднє означення процесу.

Під *процесом* розуміють абстракцію ОС, яка об'єднує все необхідне для виконання однієї програми в певний момент часу.

Програма – це деяка послідовність машинних команд, що зберігається на диску, в разі необхідності завантажується у пам'ять і виконується. Можна сказати, що під час виконання програму представляє процес.

Однозначна відповідність між програмою і процесом встановлюється тільки в конкретний момент часу: один процес у різний час може виконувати код декількох програм, код однієї програми можуть виконувати декілька процесів одночасно.

Для успішного виконання програми потрібні певні ресурси. До них належать:

- ресурси, необхідні для послідовного виконання програмного коду (передусім процесорний час);
- ресурси, що дають можливість зберігати інформацію, яка забезпечує виконання програмного коду (регістри процесора, оперативна пам'ять тощо).

Ці групи ресурсів визначають дві складові частини процесу:

- послідовність виконуваних команд процесора;
- набір адрес пам'яті (*адресний простір*), у якому розташовані ці команди і дані для них.

Виділення цих частин виправдане ще й тим, що в рамках одного адресного простору може бути кілька паралельно виконуваних послідовностей команд, що спільно використовують одні й ті ж самі дані. Необхідність розмежування послідовності команд і адресного простору підводить до поняття потоку.

Потоком (потік керування, нитка, thread) називають набір послідовно виконуваних команд процесора, які використовують загальний адресний простір процесу. Оскільки в системі може одночасно бути багато потоків, завданням ОС є організація перемикання процесора між ними і планування їхнього виконання. У багатопроцесорних системах код окремих потоків може виконуватися на окремих процесорах.

Тепер можна дати ще одне означення процесу.

Процесом називають сукупність одного або декількох потоків і захищеного адресного простору, у якому ці потоки виконуються.

Захищеність адресного простору процесу є його найважливішою характеристикою. Код і дані процесу не можуть бути прямо прочитані або перезаписані іншим процесом; у такий спосіб захищаються від багатьох програмних помилок і спроб несанкціонованого доступу. Природно, що неприпустимим є тільки *прямий* доступ (наприклад, запис у пам'ять за допомогою простої інструкції перенесення даних); обмін даними між процесами принципово можливий, але для цього мають бути використані спеціальні засоби, які називають засобами *міжпроцесової взаємодії* (див. розділ 6). Такі засоби складніші за прямий доступ і працюють повільніше, але при цьому забезпечують захист від випадкових помилок у разі доступу до даних.

На відміну від процесів *потоки розпоряджаються загальною пам'яттю*. Дані потоку не захищені від доступу до них інших потоків за умови, що всі вони виконуються в адресному просторі одного процесу. Це надає додаткові можливості для розробки застосувань, але ускладнює програмування.

Захищений адресний простір процесу задає абстракцію виконання коду на окремій машині, а потік забезпечує абстракцію послідовного виконання команд на одному виділеному процесорі.

Адресний простір процесу не завжди відповідає адресам оперативної пам'яті. Наприклад, у нього можуть відображатися файли або регістри контролерів введення-виведення, тому запис за певною адресою в цьому просторі призведе до запису у файл або до виконання операції введення-виведення. Таку технологію називають *відображенням у пам'ять* (memory mapping).

3.1.2. Моделі процесів і потоків

Максимально можлива кількість процесів (захищених адресних просторів) і потоків, які в них виконуються, може варіюватися в різних системах.

- В однозадачних системах є тільки один адресний простір, у якому в кожен момент часу може виконуватися один потік.
- У деяких вбудованих системах теж є один адресний простір (один процес), але в ньому дозволене виконання багатьох потоків. У цьому разі можна організовувати паралельні обчислення, але захист даних застосувань не реалізовано.
- У системах, подібних до традиційних версій UNIX, допускається наявність багатьох процесів, але в рамках адресного простору процесу виконується тільки один потік. Це традиційна однопотокова *модель процесів*. Поняття потоку в даній моделі не застосовують, а використовують терміни «перемикання між процесами», «планування виконання процесів», «послідовність команд процесу» тощо (тут під процесом розуміють його єдиний потік).
- У більшості сучасних ОС (таких, як лінія Windows XP, сучасні версії UNIX) може бути багато процесів, а в адресному просторі кожного процесу – багато потоків. Ці системи підтримують багатопотоковість або реалізують *модель потоків*. Процес у такій системі називають *багатопотоковим процесом*.

Надалі для позначення послідовності виконуваних команд вживатимемо термін «потік», за винятком ситуацій, коли обговорюватиметься реалізація моделі процесів.

3.1.3. Складові елементи процесів і потоків

До елементів процесу належать:

- захищений адресний простір;
- дані, спільні для всього процесу (ці дані можуть спільно використовувати всі його потоки);
- інформація про використання ресурсів (відкриті файли, мережні з'єднання тощо);
- інформація про потоки процесу.

Потік містить такі елементи:

- стан процесора (набір поточних даних із його регістрів), зокрема лічильник поточної інструкції процесора;
- стек потоку (ділянка пам'яті, де перебувають локальні змінні потоку й адреси повернення функцій, що викликані у його коді).

3.2. Багатопотоковість та її реалізація

3.2.1. Поняття паралелізму

Використання декількох потоків у застосуванні означає внесення в нього паралелізму (concurrency). *Паралелізм* – це одночасне (з погляду прикладного програміста) виконання дій різними фрагментами коду застосування. Така одночасність може бути реалізована на одному процесорі шляхом перемикання задач (випадок псевдопаралелізму), а може ґрунтуватися на паралельному виконанні коду на декількох процесорах (випадок справжнього паралелізму). Потоки абстрагують цю відмінність, даючи можливість розробляти застосування, які в однопроцесорних системах використовують псевдопаралелізм, а при доданні процесорів – справжній паралелізм (такі застосування масштабуються зі збільшенням кількості процесорів).

3.2.2. Види паралелізму

Можна виділити такі основні види паралелізму:

- паралелізм багатопроцесорних систем;
- паралелізм операцій введення-виведення;
- паралелізм взаємодії з користувачем;
- паралелізм розподілених систем.

Перший з них є справжнім паралелізмом, тому що у багатопроцесорних системах інструкції виконують декілька процесорів одночасно. Особливості використання такого паралелізму будуть розглянуті в розділі 20. Інші види паралелізму можуть виникати і в однопроцесорних системах тоді, коли для продовження виконання програмного коду необхідні певні зовнішні дії.

Паралелізм операцій введення-виведення

Під час виконання операції введення-виведення (наприклад, обміну даними із диском) низькорівневий код доступу до диска і код застосування не можуть виконуватись одночасно. У цьому разі застосуванню треба почекати завершення операції введення-виведення, звільнивши на цей час процесор. Природним вважається зайняти на цей час процесор інструкціями іншої задачі.

Багатопотокове застосування може реалізувати цей вид паралелізму через створення нових потоків, які виконуватимуться, коли поточний потік очікує операції введення-виведення. Так реалізується асинхронне введення-виведення, коли застосування продовжує своє виконання, не чекаючи на завершення операцій введення-виведення. У розділі 15 буде розглянуто організацію такого введення-виведення.

Паралелізм взаємодії з користувачем

Під час інтерактивного сеансу роботи користувач може виконувати різні дії із застосуванням (і очікувати негайної реакції на них) до завершення обробки попередніх дій. Наприклад, після запуску команди «друкування документа» він може негайно продовжити введення тексту, не чекаючи завершення друкування.

Щоб розв'язати це завдання, можна виділити окремі потоки для безпосередньої взаємодії із користувачем (наприклад, один потік може очікувати введення з клавіатури, інший — від миші, додаткові потоки — відображати інтерфейс користувача). Основні задачі застосування (розрахунки, взаємодія з базою даних тощо) у цей час виконуватимуть інші потоки.

Паралелізм розподілених застосувань

Розглянемо серверне застосування, яке очікує запити від клієнтів і виконує дії у відповідь на запит. Якщо клієнтів багато, запити можуть надходити часто, майже водночас. Якщо тривалість обробки запиту перевищує інтервал між запитами, сервер буде змушений поміщати запити в чергу, внаслідок чого знижується продуктивність. При цьому використання потоків дає можливість організувати паралельне обслуговування запитів, коли основний потік приймає запити, відразу передає їх для виконання іншим потокам і очікує нових.

3.2.3. Переваги і недоліки багатопотоковості

Назвемо проблеми, які можуть бути вирішені за допомогою потоків.

- Використання потоків дає змогу реалізувати різні види паралелізму і дозволяє застосуванню масштабуватися із ростом кількості процесорів.
- Для підтримки потоків потрібно менше ресурсів, ніж для підтримки процесів (наприклад, немає необхідності виділяти для потоків адресний простір).
- Для обміну даними між потоками може бути використана спільна пам'ять (адресний простір їхнього процесу). Це ефективніше, ніж застосовувати засоби міжпроцесової взаємодії.

Незважаючи на перелічені переваги, використання потоків не є універсальним засобом розв'язання проблем паралелізму, і пов'язане з деякими труднощами.

- Розробляти і налагоджувати багатопотокові програми складніше, ніж звичайні послідовні програми; досить часто впровадження багатопотоковості призводить до зниження надійності застосувань. Організація спільного використання адресного простору декількома потоками вимагає, щоб програміст мав високу кваліфікацію.
- Використання потоків може спричинити зниження продуктивності застосувань. Переважно це трапляється в однопроцесорних системах (наприклад, у таких системах спроба виконати складний розрахунок паралельно декількома потоками *призводить лише до зайвих витрат на перемикання між потоками, кількість* виконаних корисних інструкцій залишиться тією ж самою).

Переваги і недоліки використання потоків потрібно враховувати під час виконання будь-якого програмного проекту. Насамперед доцільно розглядати можливість розв'язати задачу в рамках моделі процесів. При цьому, однак, варто брати до уваги не лише поточні вимоги замовника, але й необхідність розвитку застосування, його масштабування тощо. Можливо, з урахуванням цих факторів використання потоків буде виправдане.

3.2.4. Способи реалізації моделі потоків

Перш ніж розглянути основні підходи до реалізації моделі потоків, дамо означення важливих понять потоку користувача і потоку ядра.

Потік користувача – це послідовність виконання команд в адресному просторі процесу. Ядро ОС не має інформації про такі потоки, вся робота з ними виконується в режимі користувача. Засоби підтримки потоків користувача надають спеціальні системні бібліотеки; вони доступні для прикладних програмістів у вигляді бібліотечних функцій. Бібліотеки підтримки потоків у наш час звичайно реалізують набір функцій, визначений стандартом POSIX (відповідний розділ стандарту називають POSIX.1b); тут прийнято говорити про підтримку *потоків POSIX*.

Потік ядра – це послідовність виконання команд в адресному просторі ядра. Потоками ядра управляє ОС, перемикання ними можливе тільки у привілейованому режимі. Є потоки ядра, які відповідають потокам користувача, і потоки, що не мають такої відповідності.

Співвідношення між двома видами потоків визначає реалізацію моделі потоків. Є кілька варіантів такої реалізації (схем багатопотоковості); розглянемо найважливіші з них (рис. 3.1).

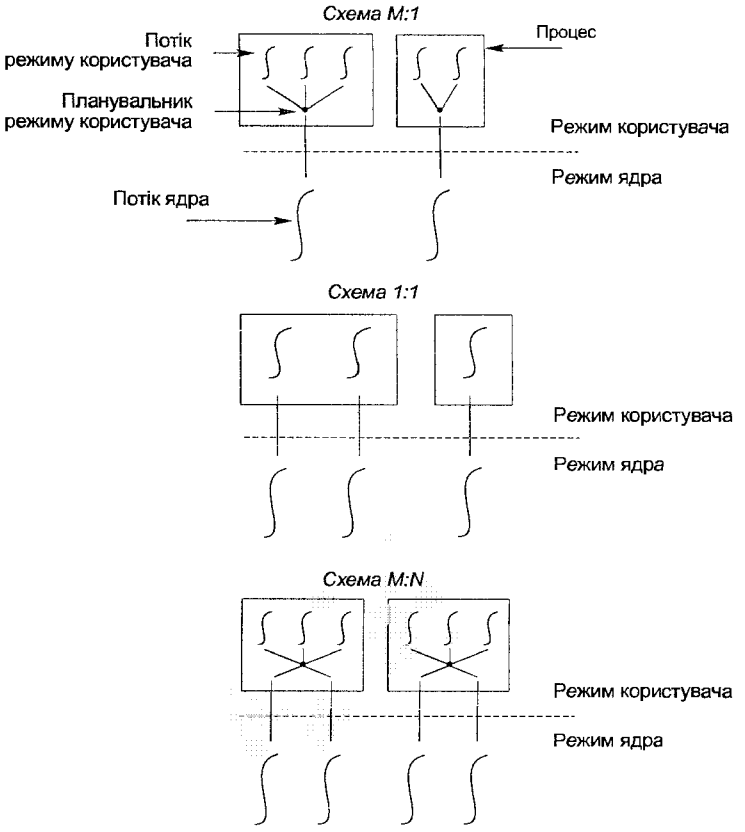

Рис. 3.1. Способи реалізації моделі потоків

Схема багатопотоковості M:1 (є найранішою) реалізує багатопотоковість виняткова в режимі користувача. При цьому кожен процес може містити багато потоків користувача, однак про наявність цих потоків ОС не відомо, вона працює тільки із процесами. За планування потоків і перемикання контексту відповідає бібліотека підтримки потоків. Схема вирізняється ефективністю керування потоками (для цього немає потреби переходити в режим ядра) і не потребує для реалізації зміни ядра ОС. Проте нині її практично не використовують через два суттєвих недоліки, що не відповідають ідеології багатопотоковості.

- Схема M:1 не дає змоги скористатися багатопроцесорними архітектурами, оскільки визначити, який саме код виконуватиметься на кожному із процесорів, може тільки ядро ОС. У результаті всі потоки одного процесу завжди виконуватимуться на одному процесорі.
- Оскільки системні виклики обробляються на рівні ядра ОС, блокувальний системний виклик (наприклад, виклик, який очікує введення даних користувачем) зупинятиме всі потоки процесу, а не лише той, що зробив цей виклик.

Схема багатопотоковості 1:1 ставить у відповідність кожному потоку користувача один потік ядра. У цьому разі планування і перемикання контексту зачіпають лише потоки ядра, у режимі користувача ці функції не реалізовані. Оскільки ядро ОС має інформацію про потоки, ця схема вільна від недоліків попередньої (різні потоки можуть виконуватися на різних процесорах, а при зупиненні одного потоку інші продовжують роботу). Вона проста і надійна в реалізації і сьогодні є найпоширенішою. Хоча схема передбачає, що під час керування потоками треба постійно перемикатися між режимами процесора, на практиці втрата продуктивності внаслідок цього виявляється незначною.

Схема багатопотоковості M:N. У цій схемі присутні як потоки ядра, так і потоки користувача, які відображаються на потоки ядра так, що один потік ядра може відповідати декільком потокам користувача. Число потоків ядра може бути змінене програмістом для досягнення максимальної продуктивності. Розподіл потоків користувача по потоках ядра виконується в режимі користувача, планування потоків ядра – у режимі ядра. Схема є складною в реалізації і сьогодні здає свої позиції схемі 1:1.

3.3. Стани процесів і потоків

Для потоку дозволені такі стани:

- *створення* (new) – потік перебуває у процесі створення;
- *виконання* (running) – інструкції потоку виконує процесор (у конкретний момент часу на одному процесорі тільки один потік може бути в такому стані);
- *очікування* (waiting) – потік очікує деякої події (наприклад, завершення операції введення-виведення); такий стан називають також заблокованим, а потік – припиненим;
- *готовність* (ready) – потік очікує, що планувальник перемкне процесор на нього, при цьому він має всі необхідні йому ресурси, крім процесорного часу;
- *завершення* (terminated) – потік завершив виконання (якщо при цьому його ресурси не були вилучені з системи, він переходить у додатковий стан – *стан зомбі*).

Можливі переходи між станами потоку зображені на рис. 3.2.

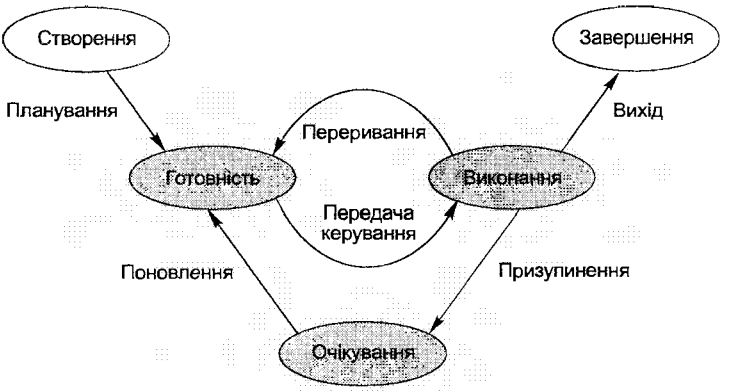

Рис. 3.2. Стани потоку

Перехід потоків між станами очікування і готовності реалізовано на основі *планування задач*, або *планування потоків*. Під час планування потоків визначають, який з потоків треба відновити після завершення операції введення-виведення, як організувати очікування подій у системі.

Для здійснення переходу потоків між станами готовності та виконання необхідне *планування процесорного часу*. На основі алгоритмів такого планування визначають, який з готових потоків потрібно виконувати в конкретний момент, коли потрібно перервати виконання потоку, щоб перемкнутися на інший готовий потік тощо. Планування задач і процесорного часу є темою розділу 4.

Відносно систем, які реалізують модель процесів, прийнято говорити про стани процесів, а не потоків, і про планування процесів; фактично стани процесу в цьому разі однозначно відповідають станам його єдиного потоку.

У багатопотокових системах також можна виділяти стани процесів. Наприклад, у багатопотоковості, реалізованій за схемою M:1, потоки змінюють свої стани в режимі користувача, а процеси — у режимі ядра.

3.4. Опис процесів і потоків

Як ми вже знаємо, одним із основних завдань операційної системи є розподіл ресурсів між процесами і потоками. Такими ресурсами є насамперед процесорний час (його розподіляють між потоками під час планування), засоби введення-виведення й оперативна пам'ять (їх розподіляють між процесами).

Для керування розподілом ресурсів ОС повинна підтримувати структури даних, які містять інформацію, що описує процеси, потоки і ресурси. До таких структур даних належать:

- таблиці розподілу ресурсів: таблиці пам'яті, таблиці введення-виведення, таблиці файлів тощо;
- таблиці процесів і таблиці потоків, де міститься інформація про процеси і потоки, присутні у системі в конкретний момент.

3.4.1. Керуючі блоки процесів і потоків

Інформацію про процеси і потоки в системі зберігають у спеціальних структурах даних, які називають керуючими блоками процесів і керуючими блоками потоків. Ці структури дуже важливі для роботи ОС, оскільки на підставі їхньої інформації система здійснює керування процесами і потоками.

Керуючий блок потоку (Thread Control Block, TCB) відповідає активному потоку, тобто тому, який перебуває у стані готовності, очікування або виконання. Цей блок може містити таку інформацію:

- ідентифікаційні дані потоку (зазвичай його унікальний ідентифікатор);
- стан процесора потоку: користувацькі регістри процесора, лічильник інструкцій, покажчик на стек;
- інформацію для планування потоків.

Таблиця потоків – це зв'язний список або масив керуючих блоків потоку. Вона розташована в захищеній області пам'яті ОС.

Керуючий блок процесу (Process Control Block, PCB) відповідає процесу, що присутній у системі. Такий блок може містити:

- ідентифікаційні дані процесу (унікальний ідентифікатор, інформацію про інші процеси, пов'язані з даним);
- інформацію про потоки, які виконуються в адресному просторі процесу (наприклад, покажчики на їхні керуючі блоки);
- інформацію, на основі якої можна визначити права процесу на використання різних ресурсів (наприклад, ідентифікатор користувача, який створив процес);
- інформацію з розподілу адресного простору процесу;
- інформацію про ресурси введення-виведення та файли, які використовує процес.

Зазначимо, що для систем, у яких реалізована модель процесів, у керуючому блоці процесу зберігають не посилання на керуючі блоки його потоків, а інформацію, необхідну безпосередньо для його виконання (лічильник інструкцій, дані для планування тощо).

Таблицю процесів організовують аналогічно до таблиці потоків. Як елементи в ній зберігатимуться керуючі блоки процесів.

3.4.2. Образи процесу і потоку

Сукупність інформації, що відображає процес у пам'яті, називають *образом процесу* (process image), а всю інформацію про потік – *образом потоку* (thread image). До образу процесу належать:

- керуючий блок процесу;
- програмний код користувача;
- дані користувача (глобальні дані програми, загальні для всіх потоків);
- інформація образів потоків процесу.

Програмний код користувача, дані користувача та інформація про потоки завантажуються в адресний простір процесу. Образ процесу звичайно не є безперервною ділянкою пам'яті, його частини можуть вивантажуватися на диск. Ці питання будуть розглянуті в розділі 9.

До образу потоку належать:

- керуючий блок потоку;
- стек ядра (стек потоку, який використовується під час виконання коду потоку в режимі ядра);
- стек користувача (стек потоку, доступний у користувацькому режимі).

Схема розташування в пам'яті образів процесу і його потоків зображена на рис. 3.3. Усі потоки конкретного процесу можуть користуватися загальною інформацією його образу.

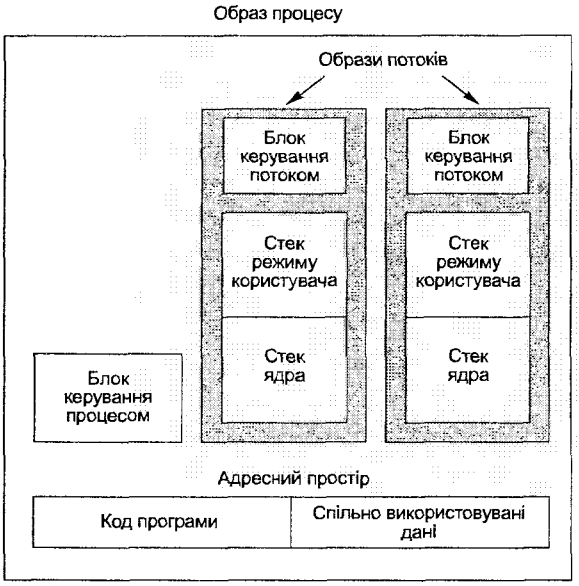

Рис. 3.3. Образи процесу і його потоків

3.5. Перемикання контексту й обробка переривань

3.5.1. Організація перемикання контексту

Найважливішим завданням операційної системи під час керування процесами і потоками є організація *перемикання контексту* — передачі керування від одного потоку до іншого зі збереженням стану процесора.

Загальних принципів перемикання контексту дотримуються у більшості систем, але їхня реалізація обумовлена конкретною архітектурою. Звичайно потрібно виконати такі операції:

- зберегти стан процесора потоку в деякій ділянці пам'яті (області зберігання стану процесора потоку);

- визначити, який потік слід виконувати наступним;
- завантажити стан процесора цього потоку із його області зберігання;
- продовжити виконання коду нового потоку.

Перемикання контексту звичайно здійснюється із залученням засобів апаратної підтримки. Можуть бути використані спеціальні регістри та ділянки пам'яті, які дають можливість зберігати інформацію про поточну задачу (коли розглядають апаратне забезпечення, аналогом поняття «потік» є поняття «задача»), а також спеціальні інструкції процесора для роботи з цими регістрами та ділянками пам'яті.

Розглянемо апаратну підтримку перемикання задач в архітектурі IA-32. Для збереження стану процесора кожної задачі (вмісту пов'язаних із нею регістрів процесора) використовують спеціальну ділянку пам'яті – сегмент стану задачі TSS. Адресу цієї області можна одержати з регістра задачі TR (це системний адресний регістр).

Для перемикання задач досить завантажити нові дані в регістр TR. У результаті значення регістрів процесора поточної задачі автоматично збережуться в її сегменті стану, після чого в регістри процесора буде завантажено стан процесора нової (або раніше перерваної) задачі й почнеться виконання її інструкцій.

Наступний потік для виконання вибирають відповідно до принципів планування потоків, які ми розглянемо в розділі 4.

3.5.2. Обробка переривань

У процесі виконання потік може бути перерваний не лише для перемикання контексту на інший потік, але й у зв'язку із програмним або апаратним перериванням (перемикання контексту теж пов'язане із перериваннями, власне, із перериванням від таймера). Із кожним перериванням надходить додаткова інформація (наприклад, його номер). На підставі цієї інформації система визначає, де буде розміщена адреса процедури оброблювача переривання (список таких адрес зберігають у спеціальній ділянці пам'яті і називають *вектором переривань*).

Наведемо приклад послідовності дій під час обробки переривання:

- збереження стану процесора потоку;
- встановлення стека оброблювача переривання;
- початок виконання оброблювача переривання (коду операційної системи); для цього з вектора переривання завантажується нове значення лічильника команд;
- відновлення стану процесора потоку після закінчення виконання оброблювача і продовження виконання потоку.

Передача керування оброблювачеві переривання, як і перемикання контексту, може відбутися практично у будь-який момент. Основна відмінність полягає в тому, що адресу, на яку передається керування, задають на основі номера переривання і зберігають у векторі переривань, а також у тому, що код оброблювача не продовжується з місця, де було перерване виконання, а починає виконуватися щораз заново.

Докладніше реалізацію обробки переривань буде розглянуто в розділі 15.

3.6. Створення і завершення процесів і потоків

Засоби створення і завершення процесів дають змогу динамічно змінювати в операційній системі набір застосувань, що виконуються. Засоби створення і завершення потоків є основою для створення багатопотокових програм.

3.6.1. Створення процесів

Базові принципи створення процесів

Процеси можуть створюватися ядром системи під час її ініціалізації. Наприклад, в UNIX-сумісних системах таким процесом може бути процес ініціалізації системи `init`, у Windows XP – процеси підсистем середовища (Win32 або POSIX). Таке створення процесів, однак, є винятком, а не правилом.

Найчастіше процеси створюються під час виконання інших процесів. У цьому разі процес, який створює інший процес, називають *предком*, а створений ним процес – *нащадком*.

Нові процеси можуть бути створені під час роботи застосування відповідно до його логіки (компілятор може створювати процеси для кожного етапу компіляції, веб-сервер – для обробки прибулих запитів) або безпосередньо за запитом користувача (наприклад, з командного інтерпретатора, графічної оболонки або файлового менеджера).

Інтерактивні та фонові процеси

Розрізняють два типи процесів з погляду їхньої взаємодії із користувачем.

- *Інтерактивні процеси* взаємодіють із користувачами безпосередньо, приймаючи від них дані, введені за допомогою клавіатури, миші тощо. Прикладом інтерактивного процесу може бути процес текстового редактора або інтегрованого середовища розробки.
- *Фонові процеси* із користувачем не взаємодіють безпосередньо. Зазвичай вони запускаються під час старту системи і чекають на запити від інших застосувань. Деякі з них (системні процеси) підтримують функціонування системи (реалізують фонове друкування, мережні засоби тощо), інші виконують спеціалізовані задачі (реалізують веб-сервери, сервери баз даних тощо). Фонові процеси також називають службами (services, у системах лінії Windows XP) або демонами (daemons, в UNIX).

3.6.2. Ієрархія процесів

Після того як процес-предок створив процес-нащадок, потрібно забезпечити їхній взаємозв'язок. Є різні варіанти розв'язання цього завдання.

Можна організувати на рівні ОС однозначний зв'язок «предок–нащадок» так, щоб для кожного процесу завжди можна було визначити його предка. Наприклад, якщо процеси визначені унікальними ідентифікаторами, то для реалізації цього підходу в керуючому блоці процесу-нащадка повинен завжди зберігатися ідентифікатор процесу-предка або посилання на його керуючий блок.

Таким чином формується ієрархія (дерево) процесів, оскільки нащадки можуть у свою чергу створювати нових нащадків і т. ін. У таких системах звичайно

існує спеціальний вихідний процес (в UNIX-системах його називають `init`), з якого починається побудова дерева процесів (його запускає ядро системи). Якщо предок завершить виконання процесу перед своїм нащадком, функції предка бере на себе вихідний процес.

З іншого боку, зв'язок «предок–нащадок» можна не реалізовувати на рівні ОС. При цьому всі процеси виявляються рівноправними. Якщо зв'язок «предок–нащадок» для конкретної пари процесів все ж таки потрібен, за його підтримку відповідають самі процеси (процес-предок, наприклад, може сам зберегти свій ідентифікатор у структурі даних нащадка у разі його створення).

Взаємозв'язок між процесами не обмежується лише відношеннями «предок–нащадок». Наприклад, у деяких ОС є поняття сесії (session). Така сесія поєднує всі процеси, створені користувачем за час інтерактивного сеансу його роботи із системою.

3.6.3. Керування адресним простором під час створення процесів

Оскільки основним елементом процесу є захищений адресний простір, дуже важливо вирішити проблему його розподілу під час створення нового процесу. Розглянемо два різні підходи.

Системні виклики fork() і exec()

У першому підході адресний простір нащадка створюють як точну копію адресного простору предка. Така операція реалізована системним викликом, який у POSIX-системах називають `fork()`.

У цьому разі копіюється не тільки адресний простір, а й лічильник команд головного потоку процесу, тому після виклику `fork()` предок і нащадок виконуватимуть ту саму інструкцію. Розробник має визначити, у якому з двох процесів перебуває керування. Це можна зробити на підставі відмінностей між кодами повернення `fork()` для предка і нащадка.

Коли створення нового процесу відбувається шляхом дублювання адресного простору предка, виникає потреба у спеціальних засобах для завантаження програмного коду в адресний простір процесу. Такі засоби реалізує системний виклик, який у POSIX-системах називають `exec()`. Як параметр для виклику `exec()` треба вказувати весь шлях до виконуваного файла програми, який буде завантажено у пам'ять.

У системах із підтримкою `fork()` для того щоб запускати на виконання програми, після виклику `fork()` потрібно негайно викликати `exec()` (це називають технологією fork+exec).

Запуск застосування одним системним викликом

Другий підхід не розділяє дублювання адресного простору і завантаження коду — ці етапи тут поєднані в один. У даному разі системний виклик запускає на виконання задане застосування (зазвичай для цього йому потрібно вказати весь шлях до виконуваного файла цього застосування).

Можна виділити два етапи виконання такого системного виклику:

- виділення пам'яті під адресний простір нового процесу (жодна інформація при цьому з адресного простору предка не копіюється);
- завантаження виконуваного коду із зазначеного файла у виділений адресний простір.

Підхід із використанням `fork()` і `exec()` є гнучкішим, бо він дає змогу в разі необхідності обмежитись якимось одним етапом запуску застосування. Сучасні ОС переважно реалізують деяку комбінацію першого та другого підходів.

Технологія копіювання під час запису

Якщо під час створення процесу за допомогою виклику `fork()` щоразу завчасно виділяти пам'ять і копіювати в неї дані адресного простору предка (стек, область глобальних змінних, програмний код), то це буде доволі трудомістким завданням, особливо для процесів, що займають багато місця в пам'яті.

Для розв'язання цієї проблеми в сучасних ОС використовують технологію *копіювання під час запису* (copy-on-write). При цьому під час виклику `fork()` дані з пам'яті предка у пам'ять нащадка не копіюють, натомість відповідні ділянки пам'яті відображають в адресний простір — як нащадка, так і предка, але при цьому позначають їх як захищені від запису.

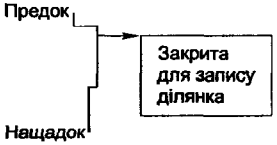

Рис. 3.4. Копіювання під час запису: пам'ять відразу після створення процесу

Як тільки нащадок або предок спробує змінити якусь із названих вище ділянок пам'яті, тобто щось записати у неї, виникає апаратне переривання. Система реагує на нього, виділяє пам'ять під нову область, відкриту для запису відповідному процесу, і змінює адресний простір для обох процесів (наприклад, для другого процесу відповідна ділянка пам'яті відкривається для запису). Слід зазначити, що в разі використання цієї технології ділянки пам'яті, які не змінюються (наприклад, ті, що містять код програми), можуть взагалі ніколи не копіюватися.

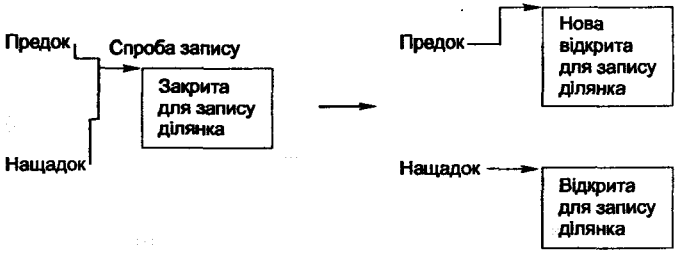

Рис. 3.5. Копіювання під час запису: пам'ять після спроби запису в неї

У багатьох сучасних системах копіювання під час запису є основною технологією керування адресним простором у разі створення процесів.

3.6.4. Особливості завершення процесів

Розглянемо три варіанти завершення процесів.

Процес *коректно завершується* самостійно після виконання своєї задачі (для інтерактивних процесів це нерідко відбувається з ініціативи користувача, який, приміром, скористався відповідним пунктом меню). Для цього код процесу має виконати системний виклик завершення процесу. Такий виклик у POSIX-системах називають `_exit()`. Він може повернути процесу, що його викликав, або ОС код повернення, який дає змогу оцінити результат виконання процесу.

Процес *аварійно завершується* через помилку. Такий вихід може бути передбачений програмістом (під час обробки помилки приймається рішення про те, що далі продовжувати виконання програми неможливо), а може бути наслідком генерування переривання (ділення на нуль, доступу до захищеної області пам'яті тощо).

Процес *завершується іншим процесом* або ядром системи. Наприклад, перед закінченням роботи ОС ядро припиняє виконання всіх процесів. Процес може припинити виконання іншого процесу за допомогою системного виклику, який у POSIX-системах називають `kill()`.

Після того як процес завершить свою роботу, пам'ять, відведена під його адресний простір, звільняється і може бути використана для інших потреб. Усі потоки цього процесу теж припиняють роботу. Якщо у даного процесу є нащадки, їхня робота переважно не припиняється слідом за роботою предка. Інтерактивні процеси звичайно завершуються у разі виходу користувача із системи.

3.6.5. Синхронне й асинхронне виконання процесів

Коли у системі з'являється новий процес, для старого процесу є два основних варіанти дій:

- продовжити виконання паралельно з новим процесом — такий режим роботи називають *асинхронним виконанням*;
- призупинити виконання доти, поки новий процес не буде завершений, — такий режим роботи називають *синхронним виконанням*. (У цьому разі використовують спеціальний системний виклик, який у POSIX-системах називають `wait()`.)

Вибір того чи іншого режиму залежить від конкретної задачі. Так, наприклад, веб-сервер може створювати процеси-нащадки для обробки запитів (якщо наявного набору нащадків недостатньо для такої обробки). У цьому разі обробка має бути асинхронною, бо відразу після створення нащадка предок має бути готовий до отримання наступного запиту. З іншого боку, компілятор C під час запуску процесів, що відповідають етапам компіляції, має чекати завершення кожного етапу, перш ніж перейти до наступного, — у такому разі використовують синхронну обробку.

3.6.6. Створення і завершення потоків

Особливості створення потоків

Процедура створення потоків має свої особливості, пов'язані насамперед із тим, що потоки створюють у рамках вже існуючого адресного простору (конкретного процесу або, можливо, ядра системи). Існує кілька ситуацій, у яких може бути створений новий потік.

Якщо процес створюється за допомогою системного виклику `fork()`, після розподілу адресного простору автоматично створюється потік усередині цього процесу (найчастіше це — головний потік застосування).

Можна створювати потоки з коду користувача за допомогою відповідного системного виклику.

У багатьох ОС є спеціальні потоки, які створює ядро системи (код ядра теж може виконуватися в потоках).

Під час створення потоку система повинна виконати такі дії.

1. Створити структури даних, які відображають потік в ОС.
2. Виділити пам'ять під стек потоку.
3. Встановити лічильник команд процесора на початок коду, який буде виконуватися в потоці; цей код називають *процедурою* або *функцією потоку*, покажчик на таку процедуру передають у системний виклик створення потоку.

Слід відзначити, що під час створення потоків, на відміну від створення процесів немає необхідності виділяти нову пам'ять під адресний простір, тому зазвичай потоки створюються швидше і з меншими затратами ресурсів.

Локальні змінні функції потоку розташовані у стеку потоку і доступні лише його коду, глобальні змінні доступні всім потокам.

Особливості завершення потоків

Під час завершення потоку його ресурси вивільняються (насамперед, стек); ця операція звичайно виконується швидше, ніж завершення процесу. Потік може бути завершений, коли керування дійде до кінця процедури потоку; є також спеціальні системні виклики, призначені для дострокового припинення виконання потоків.

Як і процеси, потоки можуть виконуватися синхронно й асинхронно. Потік, створивши інший потік, може призупинити своє виконання до його завершення. Таке очікування називають *приєднанням потоків* (thread joining, очікує той, хто приєднує). Після завершення приєднаного потоку потік, який очікував його завершення, може дістати статус виконання. Під час створення потоку можна визначити, чи підлягає він приєднанню (якщо потік не можна приєднати, його називають *від'єднаним* — detached). Якщо потік не є від'єднаним (nondetached або joinable, такий потік називатимемо *приєднуваним*), після завершення його обов'язково потрібно приєднувати, щоб уникнути витікання пам'яті.

Дамо деякі рекомендації щодо розробки багатопотокових програм.

- Не можна визначати порядок виконання потоків, не подбавши про його підтримку, тому що швидкість виконання потоків залежить від багатьох факторів, більшість із яких програміст не контролює. Наприклад, якщо запустити набір потоків усередині функції `f()` і не приєднати всі ці потоки до завершення `f()`,

деякі з них можуть не встигнути завершити своє виконання до моменту виходу з `f()`. Можливо навіть, що частина потоків не завершиться й до моменту наступного виклику функції, якщо програміст виконає його достатньо швидко.

- Для потоків не підтримується така ієрархія, як для процесів. Потік, що створив інший потік, має рівні з ним права. Розраховувати на те, що такий потік сам буде продовжувати своє виконання після завершення потоків-нащадків, немає сенсу.
- Стек потоку очищається після виходу із функції потоку. Щоб цьому запобігти, не слід, наприклад, передавати нікуди покажчики на локальні змінні такої функції. Якщо необхідні змінні, значення яких мають зберігатися між викликами функції потоку, але при цьому вони не будуть доступні іншим потокам, треба скористатися локальною пам'яттю потоку (Thread Local Storage, TLS) — певним чином організованими даними, доступ до яких здійснюється за допомогою спеціальних функцій.

3.7. Керування процесами в UNIX і Linux

3.7.1. Образ процесу

В UNIX-системах образ процесу містить такі компоненти:

- керуючий блок процесу;
- код програми, яку виконує процес;
- стек процесу, де зберігаються тимчасові дані (параметри процедур, повернені значення, локальні змінні тощо);
- глобальні дані, спільні для всього процесу.

Для кожного компонента образу процесу виділяють окрему ділянку пам'яті. У розділі 9 буде докладно розглянуто структуру різних ділянок пам'яті процесу та особливості їхнього використання.

3.7.2. Ідентифікаційна інформація та атрибути безпеки процесу

Із кожним процесом у системі пов'язана ідентифікаційна інформація.

Ідентифікатор процесу (pid) є унікальним у межах усієї системи, і його використовують для доступу до цього процесу. Ідентифікатор процесу `init` завжди дорівнює одиниці.

Ідентифікатор процесу-предка (ppid) задають під час його створення. Будь-який процес має мати доступ до цього ідентифікатора. Так в UNIX-системах обов'язково підтримується зв'язок «предок–нащадок». Якщо предок процесу *P* завершує виконання, предком цього процесу автоматично стає `init`, тому `ppid` для *P* дорівнюватиме одиниці.

Із процесом також пов'язаний набір атрибутів безпеки.

- Реальні ідентифікатори користувача і групи процесу (uid, gid) відповідають користувачеві, що запустив програму, внаслідок чого в системі з'явився відповідний процес.

- Ефективні ідентифікатори користувача і групи процесу (euid, egid) використовують у спеціальному режимі виконання процесу – виконанні з правами власника. Цей режим буде розглянуто у розділі 16.

3.7.3. Керуючий блок процесу

Розглянемо структуру керуючого блоку процесу в Linux і відмінності його реалізації у традиційних UNIX-системах [58, 95].

Керуючий блок процесу в Linux відображається структурою даних task_struct. До найважливіших полів цієї структури належать поля, що містять таку інформацію:

- ідентифікаційні дані (зокрема pid – ідентифікатор процесу);
- стан процесу (виконання, очікування тощо);
- покажчики на структури предка і нащадків;
- час створення процесу та загальний час виконання (так звані таймери процесу);
- стан процесора (вміст регістрів і лічильник інструкцій);
- атрибути безпеки процесу (uid, gid, euid, egid).

Зазначимо, що в ядрі Linux відсутня окрема структура даних для потоку, тому інформація про стан процесора міститься в керуючому блоці процесу.

Крім перелічених вище, task_struct має кілька полів спеціалізованого призначення, необхідних для різних підсистем Linux:

- відомості для обробки сигналів;
- інформація для планування процесів;
- інформація про файли і каталоги, пов'язані із процесом;
- структури даних для підсистеми керування пам'яттю.

Дані полів task_struct можуть спільно використовувати декілька процесів спеціалізованого призначення, у цьому випадку такі процеси фактично є потоками. Докладніше дане питання буде розглянуте під час вивчення реалізації багатопотоковості в Linux.

Керуючі блоки процесу зберігаються в ядрі у спеціальній структурі даних. До появи ядра версії 2.4 таку структуру називали таблицею процесів системи; це був масив, максимальна довжина якого не могла перевищувати 4 Кбайт. У ядрі версії 2.4 таблиця процесів була замінена двома динамічними структурами даних, що не мають такого обмеження на довжину:

- хеш-таблицею (де в ролі хеша виступає pid процесу), ця структура дає змогу швидко знаходити процес за його ідентифікатором;
- кільцевим двозв'язним списком, ця структура забезпечує виконання дій у циклі для всіх процесів системи.

Тепер обмеження на максимальну кількість процесів перевіряється тільки всередині реалізації функції fork() і залежить від обсягу доступної пам'яті (наприклад, є відомості [58], що у системі з 512 Мбайт пам'яті можна створити близько 32 000 процесів).

Для нащадка fork() повертає нуль, а для предка – ідентифікатор (pid) створеного процесу-нащадка. Коли з якоїсь причини нащадок не був створений, fork() поверне –1. Тому звичайний код роботи з fork() має такий вигляд:

```
pid_t pid;
if ((pid = fork()) == -1) { /* помилка, аварійне завершення */ }
if (pid == 0) {
    // це нащадок
}
else {
    // це предок
    printf ("нащадок запущений з кодом %d\n", pid);
}
```

Після створення процесу він може дістати доступ до ідентифікаційної інформації за допомогою системного виклику getpid(), який повертає ідентифікатор поточного процесу, і getppid(), що повертає ідентифікатор процесу-предка:

```
#include <unistd.h>
pid_t mypid, parent_pid;
mypid = getpid();
parent_pid = getppid();
```

3.7.5. Завершення процесу

Для завершення процесу в UNIX-системах використовують системний виклик _exit(). Розглянемо реалізацію цього системного виклику в Linux. Під час його виконання відбуваються такі дії.

1. Стан процесу стає TASK_ZOMBIE (зомбі, про цей стан див. нижче).
2. Процес повідомляє предків і нащадків про те, що він завершився (за допомогою спеціальних сигналів).
3. Звільняються ресурси, виділені під час виклику fork().
4. Планувальника повідомляють про те, що контекст можна перемикати.

У прикладних програмах для завершення процесу можна використати безпосередньо системний виклик _exit() або його пакувальник – бібліотечну функцію exit(). Ця функція закриває всі потоки процесу, коректно вивільняє всі ресурси і викликає _exit() для фактичного його завершення:

```
#include<unistd.h>
void _exit(int status); // status задає код повернення
#include<stdlib.h>
void exit(int status);  // status задає код повернення
exit (128);             // вихід з кодом повернення 128
```

Зазначимо, що краще не викликати exit() тоді, коли процес може використовувати ресурси разом з іншими процесами (наприклад, він є процесом-нащадком, який має предка, причому нащадок успадкував у предка дескриптори ресурсів). Причина полягає в тому, що в цьому разі спроба звільнити ресурси в нащадку призведе до того, що вони виявляться звільненими й у предка. Для завершення таких процесів потрібно використати _exit().

3.7.7. Запуск програми

Запуск нових програм в UNIX відокремлений від створення процесів і реалізований за допомогою системних викликів сімейства exec(). На практиці звичайно реалізують один виклик (у Linux це — execve()).

У Linux підтримують концепцію *оброблювачів двійкових форматів* (binary format handlers). Формати виконуваних файлів заздалегідь реєструють в ядрі, при цьому їхній набір може динамічно змінюватися. З кожним форматом пов'язаний оброблювач — спеціальна структура даних linux_binfmt, що містить покажчики на процедури завантаження виконуваного файла (load_binary) і динамічної бібліотеки (load_shlib) для цього формату.

Тепер зупинимося на послідовності кроків виконання execve() у Linux. Вхідними параметрами цього виклику є рядок з ім'ям виконуваного файла програми, масив аргументів командного рядка (до яких у коді процесу можна буде дістати доступ за допомогою масиву argv) і масив змінних оточення (що є парами *ім'я=значення*). Виклик execve() відбувається так.

1. Відкривають виконуваний файл програми.
2. Використовуючи вміст цього файла, ініціалізують спеціальну структуру даних bprm, що містить інформацію, необхідну для виконання файла, зокрема відомості про його двійковий формат.
3. Виконують дії для забезпечення безпеки виконання коду.
4. На базі параметрів виклику формують командний рядок і набір змінних оточення програми (це спеціальні поля структури bprm).
5. Для кожного зареєстрованого оброблювача бінарного формату викликають процедуру завантаження виконуваного файла через покажчик load_binary і перевіряють результат цього виклику:
   а) у разі коректного завантаження оброблювач починає виконувати код нової програми;
   б) якщо жоден з оброблювачів не зміг коректно завантажити код, повертають помилку.

Детальному розгляду процесу завантаження програмного коду в пам'ять і форматів виконуваних файлів буде присвячено розділ 14.

Запуск виконуваних файлів у прикладних програмах

Розглянемо приклад запуску виконуваного файла за допомогою системного виклику execve(). Синтаксис цього виклику такий:

```
#include <unistd.h>
int execve( const char *filename, // повне ім'я файлу
            const char *argv[],   // масив аргументів командного рядка
            const char *envp[]);  // масив змінних оточення
```

Останнім елементом argv і envp має бути нульовий покажчик.

Повернення з виклику буде тільки тоді, коли програму не вдалося завантажити (наприклад, файл не знайдено). Тоді буде повернено –1.

Наведемо приклад формування параметрів і виклику (зазначимо, що першим елементом argv задане ім'я виконуваної програми):

```
char *prog = "./child";
char *args[] = { "./child", "arg1", NULL };
char *env[] = { NULL };
if (execve (prog, args, env) == -1) {
    printf ("помилка під час виконання програми: %s\n", prog);
    exit (-1);
}
```

Реалізація технології fork+exec

Наведемо приклад коду, що реалізує технологію fork+exec:

```
if ((pid = fork()) < 0) exit (-1);
if (pid == 0) {  // нащадок завантажує замість себе іншу програму
    // ... завдання prog, args і env
    if (execve (prog, args, env) == -1) {
        printf("помилка під час запуску нащадка\n");
        exit (-1);
    }
}
else {
    // предок продовжує роботу (наприклад, очікує закінчення нащадка)
}
```

3.7.8. Очікування завершення процесу

Коли процес завершується, його керуючий блок не вилучається зі списку й хеша процесів негайно, а залишається там доти, поки інший процес (предок) не видалить його звідти. Якщо процес насправді в системі відсутній (він завершений), а є тільки його керуючий блок, то такий процес називають *процесом-зомбі* (zombie process).

Системний виклик waitpid()

Для вилучення із системи інформації про процес в Linux можна використати описаний вище системний виклик wait(), але частіше застосовують його більш універсальний варіант – waitpid(). Цей виклик перевіряє, чи є керуючий блок відповідного процесу в системі. Якщо він є, а процес не перебуває у стані зомбі (тобто ще виконується), то процес у разі виклику waitpid() переходить у стан очікування. Коли ж процес-нащадок завершується, предок виходить зі стану очікування і вилучає керуючий блок нащадка. Якщо предок не викличе waitpid() для нащадка, той може залишитися у стані зомбі надовго.

Синхронне виконання процесів у прикладних програмах

Розглянемо реалізацію синхронного виконання процесів на базі waitpid(). Відповідно до POSIX цей системний виклик визначається так:

```
#include<sys/wait.h>
pid_t waitpid(pid_t pid,    // pid процесу, який очікуємо
              int *status,  // інформація про статус завершення нащадка
              int options); // задаватимемо як 0
```

Параметр pid можна задавати як 0, що означатиме очікування процесу із тієї ж групи, до якої належить предок, або як –1, що означатиме очікування будь-якого нащадка. Наведемо приклад реалізації синхронного виконання з очікуванням:

```
pid_t pid;
if ((pid = fork()) == -1) exit(-1);
if (pid == 0) {
    // нащадок – виклик exec()
}
else {
    //  предок – чекати нащадка
    int status;
    waitpid (pid, &status, 0);
    // продовжувати виконання
}
```

Зі значення status можна отримати додаткову інформацію про процес-нащадок, що завершився. Для цього є низка макропідстановок з <sys/wait.h>:

- WIFEXITED(status) – ненульове значення, коли нащадок завершився нормально;
- WEXITSTATUS(status) – код повернення нащадка (тільки коли WIFEXITED() !=0).

Код повернення нащадка отриманий таким чином:

```
waitpid (pid, &status, 0);
if (WIFEXITED(status))
     printf("нащадок завершився з кодом %d\n", WEXITSTATUS(status));
```

Асинхронне виконання процесів може бути реалізоване на основі сигналів і буде розглянуте в наступному розділі.

3.7.9. Сигнали

За умов багатозадачності виникає необхідність сповіщати процеси про події, що відбуваються в системі або інших процесах. Найпростішим механізмом такого сповіщення, визначеним POSIX, є *сигнали*. Сигнали діють на кшталт програмних переривань: процес після отримання сигналу негайно реагує на нього викликом спеціальної функції – *оброблювача* цього сигналу (signal handler) або виконанням дії за замовчуванням для цього сигналу. Із кожним сигналом пов'язаний його номер, що є унікальним у системі. Жодної іншої інформації разом із сигналом передано бути не може.

Сигнали є найпростішим механізмом міжпроцесової взаємодії в UNIX-системах, але, оскільки з їхньою допомогою можна передавати обмежені дані, вони переважно використовуються для повідомлення про події.

Типи сигналів

Залежно від обставин виникнення сигнали поділяють на синхронні й асинхронні. *Синхронні сигнали* виникають під час виконання активного потоку процесу (зазвичай через помилку – доступ до невірної ділянки пам'яті, некоректну роботу із плаваючою крапкою, виконання неправильної інструкції процесора). Ці сигнали генерує ядро ОС і негайно відправляє їх процесу, потік якого викликав помилку.

Асинхронні сигнали процес може отримувати у будь-який момент виконання:

- програміст може надіслати асинхронний сигнал іншому процесу, використовуючи системний виклик, який у POSIX-системах називають `kill()`, параметрами цього виклику є номер сигналу та ідентифікатор процесу;
- причиною виникнення сигналу може бути також деяка зовнішня подія (натискання користувача на певні клавіші, завершення процесу-нащадка тощо).

Під час обробки таких сигналів система має перервати виконання поточного коду, виконати код оброблювача і повернутися до тієї ж самої інструкції, що виконувалась у момент отримання сигналу.

Диспозиція сигналів

На надходження сигналу процес може реагувати одним із трьох способів (спосіб реакції процесу на сигнал називають диспозицією сигналу):

- викликати оброблювач сигналу;
- проігнорувати сигнал, який у цьому випадку «зникне» і не виконає жодної дії;
- використати диспозицію, передбачену за замовчуванням (така диспозиція задана для кожного сигналу, найчастіше це — завершення процесу).

Процес може задавати диспозицію для кожного сигналу окремо.

Блокування сигналів

Процес може не лише задавати диспозицію сигналів, а й визначати свою готовність у цей момент приймати сигнали певного типу. Якщо процес не готовий, він може ці сигнали заблокувати. Якщо заблокований сигнал має бути доставлений процесу, система ставить його в чергу, де він залишатиметься доти, поки процес його не розблокує. Процес блокує й розблоковує сигнали шляхом зміни *маски сигналів процесу* (спеціальної структури даних, де зберігається інформація про те, які сигнали можуть бути негайно доставлені процесу, а які в цей момент заблоковані).

Маска сигналів є характеристикою процесу, зазвичай вона зберігається в його керуючому блоці. Процеси-нащадки успадковують маску сигналів предка.

Приклади сигналів

Назвемо сигнали, що визначені POSIX і підтримуються в Linux (у дужках поруч з ім'ям сигналу наведено його номер).

До синхронних сигналів належить, наприклад, сигнал `SIGSEGV` (11), який генерує система під час записування в захищену ділянку пам'яті.

До асинхронних сигналів належать:

- `SIGHUP` (1) — розрив зв'язку (наприклад, вихід користувача із системи);
- `SIGINT` і `SIGQUIT` (2, 3) — сигнали переривання програми від клавіатури (генеруються під час натискання користувачем відповідно Ctrl+C і Ctrl+\);
- `SIGKILL` (9) — негайне припинення роботи програми (для такого сигналу не можна змінювати диспозицію);
- `SIGUSR1` і `SIGUSR2` (10, 12) — сигнали користувача, які можуть використовувати прикладні програми;
- `SIGTERM` (15) — пропозиція програмі завершити її роботу (цей сигнал, на відміну від `SIGKILL`, може бути зігнорований).

Діями за замовчуванням для всіх зазначених сигналів, крім SIGSEGV, є припинення роботи програми (для SIGSEGV додатково генерується *дамп пам'яті* (core dump) – файл, у якому зберігається образ адресного простору процесу для наступного аналізу).

Задання диспозиції сигналів

Для задання диспозиції сигналу використовують системний виклик sigaction().

```
#include <signal.h>
int sigaction(int signum,              // номер сигналу
      struct sigaction *action,        // нова диспозиція
      struct sigaction *old_action);   // повернення попередньої диспозиції
```

Диспозицію описують за допомогою структури sigaction з такими полями:

- sa_handler – покажчик на функцію-оброблювач сигналу;
- sa_mask – маска сигналу, що задає, які сигнали будуть заблоковані всередині оброблювача;
- sa_flag – додаткові прапорці.

Обмежимося обнуліннням sa_mask і sa_flag (не блокуючи жодного сигналу):

```
struct sigaction action = {0};
```

Поле sa_handler має бути задане як покажчик на оголошену раніше функцію, що має такий вигляд:

```
void user_handler (int signum) {
    // обробка сигналу
}
```

Ця функція і стає оброблювачем сигналів.

Параметр signum визначає, який сигнал надійшов в оброблювач (той самий оброблювач може бути зареєстрований для декількох сигналів кількома викликами sigaction()).

Після реєстрації оброблювач викликатиметься завжди, коли надійде відповідний сигнал:

```
#include<signal.h>
void sigint_handler (int signum) {
   // обробка SIGINT
}
// ...
action.sa_handler = sigint_handler;
sigaction(SIGINT, &action, 0);
```

Якщо нам потрібно організувати очікування сигналу, то найпростішим способом є використання системного виклику pause(). У цьому разі процес переходить у режим очікування, з якого його виведе поява будь-якого сигналу:

```
// задати оброблювачі за допомогою sigaction()
pause();   // чекати сигналу
```

Генерування сигналів

Тепер зупинимося на тому, як надіслати сигнал процесу. Для цього використовують системний виклик `kill()`.

```
#include<signal.h>
int kill (pid_t pid,   // ідентифікатор процесу
          int signum); // номер сигналу
```

Наведемо приклад надсилання `SIGHUP` процесу, `pid` якого задано у командному рядку:

```
kill (atoi(argv[1]), SIGHUP);
```

Організація асинхронного виконання процесів

Розглянемо, як можна використати обробку сигналів для організації асинхронного виконання процесів. Як відомо, виклик `waitpid()` спричиняє організацію синхронного виконання нащадка. Коли необхідно запустити процес-нащадок асинхронно, то здається природним не викликати в процесі-предку `waitpid()` для цього нащадка. Але після завершення процесу-нащадка він перетвориться на процес-зомбі.

Щоб цього уникнути, необхідно скористатися тим, що під час завершення процесу-нащадка процес-предок отримує спеціальний сигнал `SIGCHLD`. Якщо в оброблювачі цього сигналу викликати `waitpid()`, то це призведе до вилучення інформації про процес-нащадок з таблиці процесів, внаслідок чого зомбі в системі не залишиться.

```
void clear_zombie(int signum) {
    waitpid(-1, NULL, 0);
}

struct sigaction action = { 0 };
action.sa_handler = clear_zombie;
sigaction(SIGCHLD, &action, 0);
if ( (pid = fork()) == -1) _exit();
if (pid == 0) {
    // ... нащадок запущений асинхронно
    _exit();
} else {
    // предок – немає виклику waitpid()
}
```

3.8. Керування потоками в Linux

3.8.1. Базова підтримка багатопотоковості

Донедавна єдиним засобом підтримки багатопотоковості в ядрі Linux був системний виклик `clone()`, який дає можливість створити новий процес на базі наявного. Цей виклик багато в чому схожий на виклик `fork()`, але має кілька особливостей.

- Для нового процесу потрібно задати спеціальний набір прапорців, що визначають, як будуть розподілятися ресурси між предком і нащадком. До ресурсів, які можна розділяти, належать адресний простір, інформація про відкриті файли, оброблювачі сигналів. Зазначимо, що у традиційній реалізації `clone()`

відсутнє спільне використання ідентифікатора процесу (pid), ідентифікатора предка та деяких інших важливих атрибутів.

- Для нового процесу потрібно задати новий стек (оскільки внаслідок виклику `clone()` два процеси можуть спільно використовувати загальну пам'ять, для них неприпустиме використання загального стека).

Підтримка `clone()` означає реалізацію багатопотоковості за схемою 1:1. При цьому первинними в системі є процеси, а не потоки (у Linux послідовності інструкцій процесора, з якими працює ядро, називають процесами, а не потоками ядра).

Традиційна реалізація багатопотоковості в Linux визначає, що потоки користувача відображаються на процеси в ядрі. Тому в цьому розділі недоцільно розглядати окремо структури даних потоку в Linux — він відображається такою ж самою структурою `task_struct`, як і процес.

Бібліотека підтримки потоків LinuxThreads

Безпосередньо використовувати системний виклик `clone()` для створення багатопотокових застосувань складно. Крім того, таке використання веде до втрати здатності до перенесення застосувань (виклик `clone()` реалізований тільки в Linux і не є частиною POSIX). Для того щоб прикладний програміст міг використати у своїх програмах потоки POSIX, необхідна реалізація цього стандарту в бібліотеці підтримки потоків.

Основною бібліотекою підтримки потоків у Linux довгий час була бібліотека LinuxThreads. У ній підтримуються ті можливості стандарту POSIX.1b, які реалізуються на основі виклику `clone()`. На жаль, через обмеженість цього виклику бібліотеці властиві об'єктивні недоліки, які ми розглянемо докладно.

Недоліки традиційної підтримки багатопотоковості в Linux

Основи підтримки багатопотоковості в ядрі Linux не зазнали принципових змін від появи системного виклику `clone()`. За цей час Linux із експериментальної системи перетворився на систему промислового рівня, в якій виконують корпоративні застосування в цілодобовому режимі з великим навантаженням.

Таке використання системи висунуло до реалізації багатопотоковості вимоги високої надійності та масштабованості. Стало очевидно, що реалізація, заснована на системному виклику `clone()`, цим вимогам не відповідає. Назвемо деякі причини цієї невідповідності.

- Оскільки потоки, створені за допомогою `clone()`, фактично були процесами, що спільно використовують ресурси, створення кожного потоку збільшувало кількість процесів у системі.
- Кожний потік у системі, будучи за своєю суттю процесом, мав власний ідентифікатор процесу (pid), що не відповідало стандарту POSIX. Крім того, якщо потік створював інший потік, між ними виникав зв'язок «предок–нащадок», тоді як всі потоки мають бути рівноправними.
- Кожне багатопотокове застосування користувача обов'язково створювало додатковий потік (потік-менеджер), який відповідав за створення та знищення потоків і перенаправляв їм сигнали. Наявність такого потоку знижувала надійність і масштабованість системи (наприклад, у разі аварійного завершення потоку-менеджера застосування залишалося у невизначеному стані).

Усі ці фактори робили багатопотоковість у Linux швидше цікавим полем для експериментів, аніж реальним засобом підвищення продуктивності й масштабованості застосувань. Розв'язати всі проблеми шляхом усунення недоліків `clone()` і LinuxThreads виявилося неможливим, потрібне було повне перероблення засобів реалізації багатопотоковості в ядрі та створення нової бібліотеки підтримки потоків.

3.8.2. Особливості нової реалізації багатопотоковості в ядрі Linux

Спроби удосконалити реалізацію багатопотоковості у ядрі системи, а також бібліотеку підтримки потоків, яка використовує нові можливості ядра, описані в літературі [67]. Нова реалізація багатопотоковості у ядрі системи має такі особливості.

- Підвищилася продуктивність операції створення і завершення потоків. Знято обмеження на загальну кількість потоків у системі. Система залишається стабільною за умов одночасного створення і завершення сотень тисяч потоків (за наявності достатнього обсягу оперативної пам'яті) [67].
- Усі потоки процесу тепер повертають один і той самий ідентифікатор (pid), крім того, зв'язок «предок–нащадок» між ними не підтримується (у створеного потоку зберігається той самий предок, що й у потоку-творця). Як процес у системі реєструють тільки початковий потік застосування.
- Реалізацію виклику `clone()` розширено таким чином, щоб зробити непотрібним потік-менеджер.

Сьогодні нові засоби інтегровані у тестову версію ядра. Доступна також оновлена реалізація бібліотеки потоків, яка дістала назву NPTL (Native POSIX Threads Library).

Бібліотека підтримки потоків NPTL

Бібліотека NPTL призначена для того, щоб, спираючись на нові можливості ядра, забезпечити повну й коректну реалізацію стандарту потоків POSIX для використання у прикладних програмах.

Основною особливістю бібліотеки NPTL є те, що в ній збереглася підтримка схеми багатопотоковості 1:1, внаслідок чого досягається простота та надійність реалізації. При цьому продуктивність і масштабованість забезпечені оновленою підтримкою багатопотоковості у ядрі. Програмний інтерфейс бібліотеки не змінився порівняно з LinuxThreads.

3.8.3. Потоки ядра Linux

Крім процесів і потоків користувача, Linux також підтримує спеціальний вид планованих об'єктів, які мають уже знайому назву – потоки ядра. Такі потоки планують як звичайні процеси і потоки, кожен з яких має свій ідентифікатор (pid). Відмінності потоків ядра від процесів і потоків користувача полягають у тому, що:

- функції потоку для них визначають у коді ядра;
- вони виконуються тільки в режимі ядра;
- для них недоступні ділянки пам'яті, виділені в режимі користувача.

Прикладом потоку ядра Linux є kswapd. Цей потік використовують під час керування пам'яттю, його роботу буде розглянуто в розділі 9.

3.8.4. Програмний інтерфейс керування потоками POSIX

Керування потоками є частиною стандарту POSIX з 1996 року, відповідний програмний інтерфейс дістав назву *потоки POSIX* (POSIX threads). Цей стандарт реалізовано у більшості сучасних UNIX-сумісних систем, у Linux його підтримка була частково реалізована в бібліотеці LinuxThreads і значно повніше – у бібліотеці NPTL.

У цьому розділі ми зупинимося на базових засобах керування потоками POSIX, відклавши розгляд засобів синхронізації до розділу 5.

Усі типи даних і функції, визначені стандартом, мають префікс `pthread_`, для їхнього використання необхідно підключати заголовний файл `pthread.h`.

Створення потоків POSIX

Щоб додати новий потік у поточний процес, у POSIX використовують функцію `pthread_create()` із таким синтаксисом:

```
#include<pthread.h>
int pthread_create(pthread_t *th, pthread_attr_t *attr,
                   void *(*thread_fun)(void *), void *arg);
```

Розглянемо параметри цієї функції:

- `th` – покажчик на заздалегідь визначену структуру типу `pthread_t`, яка далі буде передана в інші функції роботи з потоками; далі називатимемо її *дескриптором потоку* (thread handle);
- `attr` – покажчик на структуру з атрибутами потоку (для використання атрибутів за замовчуванням потрібно передавати нульовий покажчик, деякі атрибути розглянемо пізніше);
- `thread_fun` – покажчик на функцію потоку, що має описуватися як

```
void *mythread_fun(void *value) {
    // виконання коду потоку
}
```

- `arg` – дані, що передаються у функцію потоку (туди вони потраплять як параметр `value`).

Ось приклад створення потоку POSIX:

```
#include<pthread.h>
// функція потоку
void *thread_fun(void *num) {
   printf ("потік номер %d\n", (int)num);
}
// ...
pthread_t th;
// створюємо другий потік
pthread_create (&th, NULL, thread_fun, (void *)++thread_num);
// тут паралельно виконуються два потоки
```

Для потоків, створених у такий спосіб, розмір стека задають за замовчуванням. Крім того, після завершення їх обов'язково треба приєднати.

Новий потік починає виконуватися паралельно із потоком, що його створив. Наприклад, якщо всередині функції `main()` був створений потік, то виконання продовжать два потоки: початковий програми, що виконує код `main()`, і новий.

Завершення потоків POSIX

Для завершення потоку достатньо дійти до кінця його функції потоку `thread_fun()` або викликати з неї `pthread_exit()`:

```
#include<pthread.h>
void pthread_exit(void *retval);
```

Параметр `retval` задає код повернення. Наведемо приклад завершення потоку:

```
void *thread_fun(void *num) {
    printf ("потік номер %d\n", (int)num);
    pthread_exit (0);
}
```

Для коду початкового потоку програми є дві різні ситуації:

- виконання оператора `return` або виконання всіх операторів до кінця `main()` *завершує весь процес*, при цьому всі потоки теж завершують своє виконання;
- виконання функції `pthread_exit()` усередині функції `main()` завершує тільки початковий потік, ця дія не впливає на інші потоки у програмі — вони продовжуватимуть виконання, поки самі не завершаться.

Виклик функції `pthread_exit()` використовують тільки для приєднуваних потоків, оскільки він не звільняє ресурси потоку — за це відповідає той, хто приєднає потік. Насправді, якщо такий потік не буде приєднаний негайно після завершення, його інформація залишиться в системі й може бути зчитана пізніше.

Приєднання потоків POSIX

Для очікування завершення виконання потоків використовують функцію `pthread_join()`. Вона має такий синтаксис:

```
int pthread_join(pthread_t th, void **status);
```

Ця функція блокує потік, де вона була викликана, доти, поки потік, заданий дескриптором `th`, не завершить своє виконання. Потік, на який очікують, має існувати в тому самому процесі й не бути від'єднаним. Якщо `status` не є нульовим покажчиком, після виклику він вказуватиме на дані, повернені потоком (аргумент функції `pthread_exit()`). Якщо потік уже завершився до моменту виклику функції `pthread_join()`, його інформація має бути зчитана негайно.

Після приєднання потоку інформацію про нього повністю вилучають із системи, тому коли кілька потоків очікували на той самий потік, його приєднає тільки один із них, для решти буде повернено помилку (до того моменту потрібного потоку в системі вже не існуватиме). Таких ситуацій краще уникати.

Розглянемо приклад приєднання потоку. Припустимо, що треба асинхронно запустити доволі тривалий запит до бази даних, виконати паралельно з ним деяку роботу (наприклад, відобразити індикатор виконання), після чого дочекатися результатів запиту. *Пропоноване рішення зводиться до створення окремого потоку*

для виконання запиту (потоку-помічника, helper thread) і до очікування його завершення у базовому потоці. Ось приблизний програмний код:

```
struct result_data {
    // інформація з бази даних

};
void *query_fun(void *retval) {
    struct result_data db_result;
    // одержати дані з бази (search_db() тут не задаємо)
    db_result = search_db(...);
    // визначаємо результат
    *(struct result_data *)retval = db_result;
    pthread_exit(0);
}
int main () {
    pthread_t query_th;
    void *status;
    struct result_data query_res;
    // почати виконувати запит паралельно з основним кодом main()
    pthread_create(&query_th, NULL, query_fun, &query_res);
    // вчинити деякі дії паралельно з виконанням запиту
    // тут два потоки виконуються паралельно
    pthread_join(query_th, &status); // дочекатися результату запиту
    // тепер можна безпечно використати результат запиту query_res
    return 0;
}
```

Часто програмісти забувають виконувати операцію приєднання під час завершення приєднуваних потоків (появі таких помилок сприяє й те, що за замовчуванням усі потоки є приєднуваними). У підсумку ресурси таких потоків не звільняються системою, що призводить до витікання пам'яті.

Від'єднані потоки. Задання атрибутів потоків POSIX

Для того, щоб отримати від'єднаний потік, треба задати відповідні *атрибути потоку*, виконавши такі дії.

1. Описати змінну для їхнього зберігання (атрибути потоку відображає структура даних pthread_attr_t):

   ```
   pthread_attr_t attr;
   ```

2. Ініціалізувати цю змінну так:

   ```
   pthread_attr_init(&attr);
   ```

3. Задати атрибут за допомогою виклику спеціальної функції (різним атрибутам відповідають різні функції).

Для задання атрибута, який визначає, чи буде цей потік від'єднаним чи ні, використовують таку функцію:

```
pthread_attr_setdetachstate(pthread_attr_t attr, int detachstate);
```

Другим параметром може бути одне зі значень: PTHREAD_CREATE_JOINABLE (приєднуваний потік); PTHREAD_CREATE_DETACHED (від'єднаний потік).

Ось приклад задання атрибута і його використання під час створення від'єднаного потоку:

```
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&th, &attr, ...);
```

Завершуються такі потоки виходом із функції потоку.

3.9. Керування процесами у Windows XP

Поняття процесу й потоку у Windows XP чітко розмежовані. Процеси в даній системі визначають «поле діяльності» для потоків, які виконуються в їхньому адресному просторі. Серед ресурсів, з якими процес може працювати прямо, відсутній процесор — він доступний тільки потокам цього процесу. Процес, проте, може задати початкові характеристики для своїх потоків і тим самим вплинути на їхнє виконання.

3.9.1. Складові елементи процесу

Розглянемо базові складові елементи процесу.

- Адресний простір процесу складається з набору адрес віртуальної пам'яті, які він може використати. Ці адреси можуть бути пов'язані з оперативною пам'яттю, а можуть — з відображеними у пам'ять ресурсами. Адресний простір процесу недоступний іншим процесам.
- Процес володіє системними ресурсами, такими як файли, мережні з'єднання, пристрої введення-виведення, об'єкти синхронізації тощо.
- Процес містить деяку стартову інформацію для потоків, які в ньому створюватимуться. Наприклад, це інформація про базовий пріоритет і прив'язання до процесора.
- Процес має містити хоча б один потік, який система скеровує на виконання. Без потоків у Windows XP наявність процесів неможлива.

3.9.2. Структури даних процесу

Розглянемо структури даних, пов'язані із процесом у Windows XP. Зазначимо, що у роботі з цими структурами система використовує об'єктну модель. Для виконавчої системи Windows XP кожний процес зображається об'єктом-процесом виконавчої системи (executive process object); його також називають *керуючим блоком процесу* (executive process block, EPROCESS). Для ядра системи процес зображається об'єктом-процесом ядра (kernel process object), його також називають *блоком процесу ядра* (process kernel block, KPROCESS).

У режимі користувача доступним є *блок оточення процесу* (process environment block, PEB), що перебуває в адресному просторі цього процесу.

Розглянемо структури даних процесу докладніше. Зазначимо, що EPROCESS і KPROCESS, на відміну від PEB, доступні тільки із привілейованого режиму.

Керуючий блок процесу містить такі основні елементи:

- блок процесу ядра (KPROCESS);
- ідентифікаційну інформацію;
- інформацію про адресний простір процесу (її структуру розглянемо в розділі 9);
- інформацію про ресурси, доступні процесу, та обмеження на використання цих ресурсів;
- блок оточення процесу (PEB);
- інформацію для підсистеми безпеки.

До ідентифікаційної інформації належать:

- ідентифікатор процесу (pid);
- ідентифікатор процесу, що створив цей процес (незважаючи на те, що Windows XP не підтримує відносини «предок–нащадок» автоматично, вони можуть бути задані програмним шляхом, тобто нащадок може сам призначити собі предка, задавши цей ідентифікатор);
- ім'я завантаженого програмного файла.

Блок процесу ядра містить усю інформацію, що належить до потоків цього процесу:

- покажчик на ланцюжок блоків потоків ядра, де кожний блок відповідає потоку;
- базову інформацію, необхідну ядру системи для планування потоків (ця інформація буде успадкована потоками, пов'язаними із цим процесом; її буде розглянуто в розділі 4).

Блок оточення процесу містить інформацію про процес, яка призначена для доступу з режиму користувача:

- початкову адресу ділянки пам'яті, куди завантажився програмний файл;
- покажчик на динамічну ділянку пам'яті, доступну процесу.

Цю інформацію може використати завантажувач програм або процес підсистеми Win32.

3.9.3. Створення процесів

У Win32 API прийнято модель запуску застосування за допомогою одного виклику, який створює адресний простір процесу і завантажує в нього виконуваний файл. Окремо функціональність `fork()` і `exec()` у цьому API не реалізована.

Такий виклик реалізує функція `CreateProcess()`. Вона не є системним викликом ОС – це бібліотечна функція Win32 API, реалізована в усіх Win32-сумісних системах.

Функція `CreateProcess()` потребує задання 10 параметрів, докладно їх буде розглянуто в розділі 3.9.6. Зазначимо, що системні виклики UNIX/POSIX потребують меншої кількості параметрів (як уже зазначалося, `fork()` не використовує жодного параметра, а `exec()` – використовує три параметри).

Наведемо основні кроки створення нового процесу із використанням функції `CreateProcess()`.

1. Відкривають виконуваний файл, що його ім'я задане як параметр. При цьому ОС визначає, до якої підсистеми середовища він належить. Коли це виконуваний

файл Win32, то його використовують прямо, для інших підсистем відшукують необхідний файл підтримки (наприклад, процес підсистеми POSIX для POSIX-застосувань).

2. Створюють об'єкт-процес у виконавчій системі Windows XP. При цьому виконують такі дії:
   а) створюють та ініціалізують структури даних процесу (блоки EPROCESS, KPROCESS, PEB);
   б) створюють початковий адресний простір процесу (роботу з адресним простором процесу розглянемо в розділі 9);
   в) блок процесу поміщають у кінець списку активних процесів, які підтримує система.
3. Створюють початковий потік процесу. Послідовність дій під час створення потоку розглядатимемо під час вивчення підтримки потоків у Windows XP.
4. Після створення початкового потоку підсистемі Win32 повідомляють про новий процес і його початковий потік. Це повідомлення містить їхні дескриптори (handles) – унікальні числові значення, що ідентифікують процес і потік для засобів режиму користувача. Підсистема Win32 виконує низку дій після отримання цього повідомлення (наприклад, задає пріоритет за замовчуванням) і поміщає дескриптори у свої власні таблиці процесів і потоків.
5. Після надсилання повідомлення розпочинають виконання початкового потоку (якщо він не був заданий із прапорцем відкладеного виконання).
6. Завершують ініціалізацію адресного простору процесу (наприклад, завантажують необхідні динамічні бібліотеки), після чого починають виконання завантаженого програмного коду.

3.9.4. Завершення процесів

У разі завершення процесу відповідний об'єкт-процес стає кандидатом на вилучення із системи. При цьому диспетчер об'єктів викликає метод delete для об'єктів-процесів, який закриває всі дескриптори в таблиці об'єктів цього процесу.

3.9.5. Процеси і ресурси. Таблиця об'єктів процесу

Кожен процес, як було показано в розділі 2, може користуватися ресурсами через дескриптори відповідних об'єктів. Відкриті дескриптори об'єктів є індексами в *таблиці об'єктів* (object table), що зберігається в керуючому блоці процесу. Ця таблиця містить покажчики на всі об'єкти, дескриптори яких відкриті процесом. Процес може отримати дескриптор об'єкта кількома способами:

- створивши новий об'єкт;
- відкривши дескриптор наявного об'єкта;
- успадкувавши дескриптор від іншого процесу;
- отримавши дублікат дескриптора з іншого процесу.

Кожен елемент таблиці об'єктів містить права доступу відповідного дескриптора і його режим спадкування, який визначає, чи отримають процеси, створені

розглядуваним процесом, копію дескриптора відповідного об'єкта. Режим спадкування задають під час створення об'єкта.

Об'єкт може одночасно бути використаний декількома процесами, при цьому кожен з них отримує унікальний дескриптор, що відповідає цьому об'єкту.

3.9.6. Програмний інтерфейс керування процесами Win32 API

У цьому розділі ми вперше торкнемося практичних особливостей програмування у Win32 API [31, 50]. Перш ніж перейти до основного матеріалу, зробимо кілька зауважень.

Насамперед слід звернути увагу на систему типів Win32 API. Розробники цього API для визначення типів широко застосовували синоніми імен типів, тому потрібно вміти знаходити в типах Win32 API традиційні типи мови C. Виділимо деякі базові типи:

- BOOL — його використовують для зберігання логічного значення, насправді він є цілочисловим;
- DWORD — двобайтовий цілочисловий тип без знака, аналог unsigned int;
- HANDLE — цілочисловий дескриптор об'єкта;
- LPTSTR — покажчик на рядок, що складається із двобайтових або однобайтових символів (залежно від режиму компіляції програми — із підтримкою Unicode або без неї), аналог char * або wchar_t *;
- LPCTSTR — покажчик на константний рядок, аналог const char * або const wchar_t *.

Взагалі для створення імені типу покажчика потрібно додати до імені базового типу префікс LP. Таке утворення імен траплятиметься й далі (наприклад, LPVOID означає void *, LPSECURITY_ATTRIBUTES — покажчик на структуру SECURITY_ATTRIBUTES).

Для використання засобів Win32 API у більшості випадків достатньо підключити заголовний файл windows.h. Надалі підключення цього файла матиметься на увазі за замовчуванням.

Для закриття дескрипторів об'єктів буде використана API-функція CloseHandle().

Створення процесів у Win32 API

Як ми вже зазначали, для створення нового процесу у Win32 використовують функцію CreateProcess().

```
BOOL CreateProcess ( LPCTSTR app_name, LPCTSTR cmd_line,
    LPSECURITY_ATTRIBUTES psa_proc, LPSECURITY_ATTRIBUTES psa_thr,
    BOOL inherit_handles, DWORD flag_create,
    LPVOID environ, LPTSTR cur_dir,
    LPSTARTUPINFO startup_info, LPPROCESS_INFORMATION process_info );
```

де: app_name — весь шлях до виконуваного файла (NULL — ім'я виконуваного файла можна отримати з другого аргументу):

```
CreateProcess("C:/winnt/notepad.exe", ...);
```

cmd_line — повний командний рядок для запуску виконуваного файла, можливо, із параметрами (цей рядок виконується, якщо app_name дорівнює NULL, зазвичай так запускати процес зручніше):

```
CreateProcess(NULL, "C:/winnt/notepad test.txt", ...);
```

psa_proc, psa_thr — атрибути безпеки для всього процесу і для головного потоку (особливості їхнього задання розглянемо в розділах 11 і 18, а доти як значення всіх параметрів типу LPSECURITY_ATTRIBUTES задаватимемо NULL, що означає задання атрибутів безпеки за замовчуванням);

inherit_handles — керує спадкуванням нащадками дескрипторів об'єктів, які використовуються у процесі (питання спадкування дескрипторів будуть розглянуті в розділах 11 і 13);

flag_create — маска прапорців, які керують створенням нового процесу (наприклад, прапорець CREATE_NEW_CONSOLE означає, що процес запускається в новому консольному вікні);

environ — покажчик на пам'ять із новими змінними оточення, які предок може задавати для нащадка (NULL — нащадок успадковує змінні оточення предка);

cur_dir — рядок із новим значенням поточного каталогу для нащадка (NULL — нащадок успадковує поточний каталог предка);

startup_info — покажчик на заздалегідь визначену структуру даних типу STARTUPINFO, на базі якої задають параметри для процесу-нащадка;

process_info — покажчик на заздалегідь визначену структуру даних PROCESS_INFORMATION, яку заповнює ОС під час виклику CreateProcess().

Серед полів структури STARTUPINFO можна виділити:

- cb — розмір структури у байтах (це її перше за порядком поле). Звичайно перед заповненням всю структуру обнуляють, задаючи тільки cb:

```
STARTUPINFO si = { sizeof(si) };
```

- lpTitle — рядок заголовка вікна для нової консолі:

```
// ... si обнуляється
si.lpTitle = "Мій процес-нащадок";
```

Структура PROCESS_INFORMATION містить чотири поля:

- hProcess — дескриптор створеного процесу;
- hThread — дескриптор його головного потоку;
- dwProcessId — ідентифікатор процесу (process id, pid);
- dwThreadId — ідентифікатор головного потоку (thread id, tid).

Ідентифікатор pid унікально визначає процес на рівні ОС. ОС повторно використовує pid уже завершених процесів, тому небажано запам'ятовувати їхнє значення, якщо процес уже завершився або закінчився помилкою.

CreateProcess() повертає нуль, якщо під час запуску процесу сталася помилка.

Наведемо приклад виклику CreateProcess(), у якому вказані значення всіх необхідних параметрів:

```
// ... задається si
PROCESS_INFORMATION pi;
CreateProcess (NULL, " C:/winnt/notepad test.txt", NULL, NULL, TRUE,
   CREATE_NEW_CONSOLE, NULL, "D:/", &si, &pi);
printf ("pid=%d, tid=%d\n", pi.dwProcessId, pi.dwThreadId);
CloseHandle(pi.hThread);
CloseHandle(pi.hProcess);
```

Після створення процесу може виникнути необхідність змінити його характеристики з коду, який він виконує. Для цього треба отримати доступ до дескриптора поточного процесу за допомогою функції GetCurrentProcess():

```
HANDLE curph = GetCurrentProcess();
```

Ідентифікатор поточного процесу можна отримати за допомогою функції GetCurrentProcessId():

```
int pid = GetCurrentProcessId();
```

Завершення процесів у Win32 API

Для завершення процесів використовують функцію ExitProcess():

```
VOID ExitProcess (UINT exitcode);
```

де exitcode — код повернення процесу. Наприклад

```
ExitProcess (100); // вихід з кодом 100
```

Для завершення іншого процесу використовують функцію TerminateProcess():

```
BOOL TerminateProcess (HANDLE hProcess, UINT exitcode);
```

У процесі, який запустив інший процес, можна отримати код завершення цього процесу за допомогою функції

```
GetExitCodeProcess (HANDLE hProcess, LPDWORD pexit_code);
```

Тут pexit_code — покажчик на змінну, в яку заносять код завершення.

```
int exitcode;
GetExitCodeProcess(pi.hProcess, &exitcode);
printf ("Код повернення =%d\n", exitcode);
```

Синхронне й асинхронне виконання процесів у Win32 API

Для того щоб реалізувати синхронне виконання, після успішного виконання CreateProcess() процес-предок має викликати функцію очікування закінчення нащадка. Це WaitForSingleObject(), стандартна функція очікування зміни стану об'єкта Win32 API.

```
DWORD WaitForSingleObject (HANDLE ph, DWORD timeout);
```

Тут ph — дескриптор нащадка; timeout — максимальний час очікування в мілісекундах (INFINITE — необмежено).

Повернене значення може бути WAIT_FAILED через помилку.

```
BOOL res;
if (res = CreateProcess(... , &pi)) {
   CloseHandle(pi.hThread);
   if (WaitForSingleObject(pi.hProcess, INFINITE) != WAIT_FAILED)
      GetExitCodeProcess(pi.hProcess, &exitcode);
   }
   CloseHandle(pi.hProcess);
}
```

Для асинхронного виконання достатньо відразу ж закрити обидва дескриптори і не викликати WaitForSingleObject():

```
if (res = CreateProcess(... , &pi)) {
   CloseHandle(pi.hThread);
   CloseHandle(pi.hProcess);
}
```

3.10. Керування потоками у Windows XP

Для того щоб виконувати код, у рамках процесу обов'язково необхідно створити потік. У системі Windows XP реалізована модель потоків «у чистому вигляді». Процеси і потоки є різними сутностями в системі, що перебувають у чітко визначеному взаємозв'язку один з одним; для роботи з ними використовують різні системні виклики. У Windows XP ніколи не використовували модель процесів, подібну до традиційної моделі UNIX.

Багатопотоковість Windows XP базується на схемі 1:1. Кожному потоку користувача відповідає сутність у ядрі, при цьому ядро відповідає за планування потоків. Процеси не плануються.

3.10.1. Складові елементи потоку

Потік у Windows XP складається з таких елементів:

- вмісту набору регістрів, який визначає стан процесора;
- двох стеків — один використовують для роботи в режимі користувача, інший — у режимі ядра; ці стеки розміщені в адресному просторі процесу, що створив цей потік;
- локальної пам'яті потоку (TLS);
- унікального ідентифікатора потоку (thread id, tid), який вибирають із того самого простору імен, що й ідентифікатори процесів.

Сукупність стану процесора, стеків і локальної пам'яті потоку становить *контекст потоку*. Кожний потік має власний контекст. Усі інші ресурси процесу (його адресний простір, відкриті файли тощо) спільно використовуються потоками.

Розрізняють два види потоків: потоки користувача і потоки ядра, які у Windows XP називають системними робочими потоками — system worker threads. Перші з них створюють у режимі користувача й тільки за необхідності перемика-

ють у режим ядра. Інші створюють в ядрі під час його ініціалізації і виконують у режимі ядра протягом усього часу їхнього існування.

3.10.2. Структури даних потоку

Відображення потоків у системі, як і відображення процесів, засноване на об'єктній моделі Windows XP. Для виконавчої системи Windows XP кожен потік відображається об'єктом-потоком виконавчої *системи* (executive thread object), який також називають *керуючим блоком потоку* (executive thread block, ETHREAD). Для ядра системи потік відображається об'єктом-потоком ядра (kernel thread object), який також називають *блоком потоку ядра* (thread kernel block, KTHREAD).

У режимі користувача доступним є *блок оточення потоку* (thread environment block, TEB), який перебуває в адресному просторі процесу, що створив потік.

Неважко помітити, що кожній структурі даних потоку відповідає структура даних процесу (блоки EPROCESS, KPROCESS і PEB).

Керуючий блок потоку містить базову інформацію про потік, зокрема:

- блок потоку ядра;
- ідентифікатор процесу, до якого належить потік, і покажчик на керуючий блок цього процесу (EPROCESS);
- стартову адресу потоку, з якої почнеться виконання його коду;
- інформацію для підсистеми безпеки.

Блок потоку ядра, у свою чергу, містить інформацію, необхідну ядру для організації планування і синхронізації потоків, зокрема:

- покажчик на стек ядра;
- інформацію для планувальника;
- інформацію, необхідну для синхронізації цього потоку;
- покажчик на блок оточення потоку.

Інформацію для планувальника буде розглянуто в розділі 4, об'єкти синхронізації – у розділі 5.

Блок оточення потоку містить інформацію про потік, доступну для застосувань режиму користувача. До неї належать:

- ідентифікатор потоку;
- покажчик на стек режиму користувача;
- покажчик на блок оточення процесу, до якого належить потік;
- покажчик на локальну пам'ять потоку.

3.10.3. Створення потоків

Основним засобом створення потоків у Windows XP є функція `CreateThread()` Win32 API. Назвемо етапи виконання цієї функції.

1. В адресному просторі процесу створюють стек режиму користувача для потоку.
2. Ініціалізують апаратний контекст потоку (у процесор завантажують дані, що визначають його стан). Цей крок залежить від архітектури процесора.

3. Створюють об'єкт-потік виконавчої системи у призупиненому стані, для чого в режимі ядра:
   а) створюють та ініціалізують структури даних потоку (блоки ETHREAD, KTHREAD, TEB);
   б) задають стартову адресу потоку (використовуючи передану як параметр адресу процедури потоку);
   в) задають інформацію для підсистеми безпеки та ідентифікатор потоку;
   г) виділяють місце під стек потоку ядра.
4. Підсистемі Win32 повідомляють про створення нового потоку.
5. Дескриптор та ідентифікатор потоку повертають у процес, що ініціював створення потоку (викликав `CreateThread()`).
6. Починають виконання потоку (виконують перехід за стартовою адресою).

3.10.4. Особливості програмного інтерфейсу потоків

Програмний інтерфейс керування потоками у Windows XP є частиною Win32 API. Такий інтерфейс ще називають інтерфейсом *потоків Win32*. Розглянемо особливості його використання.

Створення потоків у Win32 API

У Win32 API, як зазначалося раніше, для створення потоку призначена функція `CreateThread()`, а для його завершення – `EndThread()`.

На практиці, однак, пару `CreateThread()`/`EndThread()` є сенс використати лише тоді, коли з коду, що виконує потік, не викликаються функції стандартної бібліотеки мови С (такі, як `printf()` або `strcmp()`).

Річ у тому, що функції стандартної бібліотеки С у Win32 API не пристосовані до використання за умов багатопотоковості, і для того щоб підготувати потік до роботи за таких умов, необхідно під час його створення і завершення виконувати деякі додаткові дії. Ці дії враховані у спеціальних бібліотечних функціях роботи з потоками, описаних у заголовному файлі `process.h`. Це функція `_beginthreadex()` для створення потоку й `_endthreadex()` – для завершення потоку.

Розглянемо синтаксис функції `_beginthreadex()`. Відразу ж наголосимо, що той самий набір параметрів (відмінний лише за типами) передають і у функцію `CreateThread()`.

```
#include <process.h>
unsigned long _beginthreadex( void *security, unsigned stack_size,
  unsigned WINAPI (*thread_fun)(void *),
  void *argument, unsigned init_state, unsigned *tid );
```

де: `security` – атрибути безпеки цього потоку (`NULL` – атрибути безпеки за замовчуванням);

`stack_size` – розмір стека для потоку (зазвичай 0, у цьому разі розмір буде таким самим, що й у потоку, який викликає `_beginthreadex()`);

`thread_fun` – покажчик на функцію потоку;

`argument` – додаткові дані для передачі у функцію потоку;

init_state — початковий стан потоку під час створення (0 для потоку, що почне виконуватися негайно, CREATE_SUSPEND для припиненого);

tid — покажчик на змінну, в яку буде записано ідентифікатор потоку після виклику (0, якщо цей ідентифікатор не потрібний).

Функція _beginthreadex() повертає дескриптор створеного потоку, який потрібно перетворити в тип HANDLE:

```
HANDLE th = (HANDLE)_beginthreadex( ... );
```

Після отримання дескриптора, якщо він у цій функції більше не потрібний, його закривають за допомогою CloseHandle() аналогічно до дескриптора процесу:

```
CloseHandle(th);
```

Розглянемо приклад задання функції потоку. Додаткові дані, які передаються під час виклику _beginthreadex() за допомогою параметра argument, доступні в цій функції через параметр типу void *.

```
unsigned int WINAPI thread_fun (void *num) {
    printf ("потік %d почав виконання\n", (int)num);
    // код функції потоку
    printf ("потік %d завершив виконання\n", (int)num);
}
```

Ось приклад виклику _beginthreadex() з усіма параметрами:

```
unsigned tid;
int number = 0;
HANDLE th = (HANDLE) _beginthreadex (
                    NULL, 0, thread_fun, (void *)++number, 0, &tid);
```

Після створення потоку може виникнути потреба змінити його характеристики. Якщо це необхідно зробити у функції потоку, варто знати, як отримати доступ до його дескриптора. Для цього використовують функцію GetCurrentThread():

```
unsigned int WINAPI thread_fun (void *num) {
    HANDLE curth = GetCurrentThread();
}
```

Завершення потоків у Win32 API

Функцію потоку можна завершити двома способами.

1. Виконати у ній звичайний оператор return (цей спосіб є найнадійнішим):

```
unsigned WINAPI thread_fun (void *num) {
    return 0;
}
```

2. Викликати функцію _endthreadex() з параметром, що дорівнює коду повернення:

```
unsigned WINAPI thread_fun (void *num) {
    _endthreadex(0);
}
```

Приєднання потоків у Win32 API

Приєднання потоків у Win32 API, подібно до очікування завершення процесів, здійснюється за допомогою функції `WaitForSingleObject()`. Базовий синтаксис її використання з потоками такий:

```
HANDLE th = (HANDLE) _beginthreadex (...);
if (WaitForSingleObject(th, INFINITE) != WAIT_FAILED) {
    // потік завершений успішно
}
CloseHandle(th);
```

Висновки

- Процеси і потоки є активними ресурсами обчислювальних систем, які реалізують виконання програмного коду. Потоком називають набір послідовно виконуваних команд процесора. Процес є сукупністю одного або декількох потоків і захищеного адресного простору, в якому вони виконуються. Потоки одного процесу можуть разом використовувати спільні дані, для потоків різних процесів без використання спеціальних засобів це неможливо. У традиційних системах кожний процес міг виконувати тільки один потік, виконання програмного коду пов'язували із процесами. Сучасні ОС підтримують концепцію багатопотоковості.
- Використання потоків у застосуванні означає внесення в нього паралелізму – можливості одночасного виконання дій різними фрагментами коду. Паралелізм у програмах відображає асинхронний характер навколишнього світу, його джерелами є виконання коду на декількох процесорах, операції введення-виведення, взаємодія з користувачем, запити застосувань-клієнтів. Багатопотоковість у застосуваннях дає змогу природно реалізувати цей паралелізм і домогтися високої ефективності. З іншого боку, використання багатопотоковості досить складне і вимагає високої кваліфікації розробника.
- Розрізняють потоки користувача, які виконуються в режимі користувача в адресному просторі процесу, і потоки ядра, з якими працює ядро ОС. Взаємовідношення між ними визначають схему реалізації моделі потоків. На практиці найчастіше використовують схему 1:1, коли кожному потоку користувача відповідає один потік ядра, і саме ядро відповідає за керування потоками користувача.
- Потік може перебувати в різних станах (виконання, очікування, готовності тощо). Принципи переходу з одного стану в інший залежать від принципів планування потоків і планування процесорного часу. Перехід процесу зі стану виконання у будь-який інший стан зводиться до перемикання контексту – передачі керування від одного потоку до іншого зі збереженням стану процесора.
- Кожному процесу і потоку в системі відповідає його керуючий блок – структура даних, що містить усю необхідну інформацію. Під час створення процесу (зазвичай за допомогою системного виклику `fork()`) створюють його керуючий блок, виділяють пам'ять і запускають основний потік. Створення потоку простіше і виконується швидше, оскільки не потрібно виділяти пам'ять під новий адресний простір.

Контрольні запитання та завдання

Усі практичні завдання вимагають розробки консольних застосувань, які отримують вихідні дані зі стандартного пристрою введення та подають результат на стандартний пристрій виведення з використанням засобів бібліотеки відповідної мови програмування. Для мови C такі засоби описані у заголовному файлі stdio.h, для мови C++ – у файлі iostream.

1. У чому основна перевага схеми підтримки потоків 1:1 порівняно з іншими схемами? Чому розробники ОС не відразу змогли її оцінити?
2. У яких ситуаціях під час розробки програмного забезпечення доцільніше використовувати модель процесів, а не модель потоків?
3. Для трьох станів потоків – виконання, готовності й очікування – перелічіть усі можливі переходи зі стану в стан і назвіть причини таких переходів. Скажіть також, які переходи неможливі, та поясніть чому. У якому зі станів потоки перебувають найдовше?
4. У чому полягає головний недолік реалізації таблиці процесів у вигляді масиву? Які альтернативні варіанти її реалізації можна запропонувати?
5. Скільки копій змінної var буде створено у разі виконання цього фрагмента коду? Яких значень буде надано цим копіям?

```
int main() {
   int pid = fork();
   int var = 5;
   if (pid == 0) var += 5;
   else {
      pid = fork();
      var += 10;
      if (pid) var += 5;
   }
}
```

6. Якими будуть результати виконання таких двох програм?

```
int main() {  // (1)
    int var = 5;
    if( fork() ) wait(&var);
    var++;
    printf("%d\n", var);
    return var;
}
```

```
int main() {  // (2)
    int var = 5;
    if ( fork() ) wait(&var);
    else exit(var);
    var++;
    printf("%d\n", var);
    return var;
}
```

7. Після одержання від браузера запиту веб-сервер створює новий процес для його обслуговування і продовжує очікувати наступних запитів. Розробник сервера виявив, що після обробки кожного запиту в системі залишається процес-зомбі. Дайте відповіді на такі запитання:
   а) у чому причина появи такого процесу і наскільки серйозна ця проблема з погляду витрат пам'яті?
   б) як виправити код сервера, щоб процеси-зомбі не виникали?

8. Напишіть функцію, виклик якої призведе до знищення всіх процесів-зомбі, створених поточним процесом.
9. Розробіть простий командний інтерпретатор для Linux і Windows XP. Він повинен видавати підказку (наприклад, «>»), обробляти введений користувачем командний рядок (що містить ім'я виконуваного файлу програми та її аргументи) і запускати задану програму. Асинхронний запуск здійснюють уведенням «&» як останнього символу командного рядка. У разі завершення командного рядка будь-яким іншим символом програма запускається синхронно. Інтерпретатор завершує роботу після введення рядка «exit». Виконання програм, запущених інтерпретатором, може бути перерване натисканням клавіш Ctrl+C, однак воно не повинне переривати виконання інтерпретатора. Для запуску програмного коду в Linux рекомендовано використовувати функцію execvp(), що приймає два параметри prog і args, аналогічні до перших двох параметрів функції execve(), і використовує змінну оточення PATH для пошуку шляху до виконуваних файлів.
10. Розробіть застосування для Linux і Windows XP, що реалізує паралельне виконання коду двома потоками. Основний потік застосування T створює потік t. Далі кожен із потоків виконує цикл (наприклад, до 30). На кожній ітерації циклу він збільшує значення локального лічильника на одиницю, відображає це значення з нового рядка і призупиняється на деякий час (потік T – на час w_T, потік t – w_t). Після завершення циклу потік T приєднує t. Як залежать результати виконання цього застосування від значень w_T і w_t? Як зміняться ці результати, якщо потік t не буде приєднано?

Розділ 4

Планування процесів і потоків

- Загальні принципи, види та стратегії планування
- Алгоритми планування
- Планування в Linux
- Планування у Windows XP

Можливість паралельного виконання потоків залежить від кількості доступних процесорів. Якщо процесор один, паралельне виконання неможливе принципово (у кожен момент часу може виконуватися тільки один потік). Якщо кількість процесорів $N > 1$, паралельне виконання може бути реалізоване тільки для N потоків (по одному потокові на процесор).

Якщо потоків у системі більше, ніж доступних процесорів, ОС повинна розв'язувати задачу *планування* (scheduling). Головна мета планування для однопроцесорної системи полягає у такій організації виконання кількох потоків на одному процесорі, за якої у користувача системи виникало б враження, що вони виконуються одночасно.

Це означення може бути розширене на багатопроцесорні системи у разі виникнення задачі планування, коли кількість потоків перевищує кількість доступних процесорів.

У цьому розділі мова йде про основні види планування, їхні принципи та алгоритми.

4.1. Загальні принципи планування

Розглянемо загальні принципи, що лежать в основі планування.

4.1.1. Особливості виконання потоків

З погляду планування виконання потоку можна зобразити як цикл чергування періодів обчислень (використання процесора) і періодів очікування введення-виведення. Інтервал часу, упродовж якого потік виконує тільки інструкції процесора, називають інтервалом використання процесора (CPU burst), інтервал часу, коли потік очікує введення-виведення, – інтервалом введення-виведення (I/O burst). Найчастіше ці інтервали мають довжину від 2 до 8 мс.

Потоки, які більше часу витрачають на обчислення і менше – на введення-виведення, називають *обмеженими можливостями процесора* (CPU bound). Вони активно використовують процесор. Основною їхньою характеристикою є час, витрачений на обчислення, інтервали використання процесора для них довші. Потоки, які більшу частину часу перебувають в очікуванні введення-виведення, називають обмеженими можливостями введення-виведення (I/O bound). Такі потоки завантажують процесор значно менше, а середня довжина інтервалу використання процесора для них невелика. Що вища тактова частота процесора, то більше потоків можна віднести до другої категорії.

Рис. 4.1. Класифікація потоків з погляду планування

4.1.2. Механізми і політика планування

Слід розрізняти *механізми* і *політику* планування. До механізмів планування належать засоби перемикання контексту, засоби синхронізації потоків тощо, до політики планування – засоби визначення моменту часу, коли необхідно перемкнути контекст. Ту частину системи, яка відповідає за політику планування, називають *планувальником* (scheduler), а алгоритм, що використовують при цьому, – *алгоритмом планування* (scheduling algorithm).

Є різні критерії оцінки політики планування, одні з них застосовні для всіх систем, інші – лише для пакетних систем або лише для інтерактивних.

Сьогодні найчастіше використовують три критерії оцінки досягнення мети.

- *Мінімальний час відгуку.* Це найважливіший критерій для інтерактивних систем. Під часом відгуку розуміють час між запуском потоку (або введенням користувачем інтерактивної команди) і отриманням першої відповіді. Для сучасних систем прийнятним часом відгуку вважають 50–150 мс.
- *Максимальна пропускна здатність.* Це кількість задач, які система може виконувати за одиницю часу (наприклад, за секунду). Такий критерій доцільно застосовувати у пакетних системах; в інтерактивних системах він може бути використаний для фонових задач. Щоб підвищити пропускну здатність, необхідно:
  - скорочувати час даремного навантаження (наприклад, час, необхідний для перемикання контексту);
  - ефективніше використати ресурси (для того, щоб ані процесор, ані пристрої введення-виведення не простоювали).

- Третім критерієм є *справедливість*, яка полягає в тому, що процесорний час потокам виділяють відповідно до їхньої важливості. Справедливість забезпечує такий розподіл процесорного часу, що всі потоки просуваються у своєму виконанні, і жоден не простоює. Відзначимо, що реалізація справедливої політики планування не завжди призводить до зменшення середнього часу відгуку. Іноді для цього потрібно зробити систему менш справедливою.

4.1.3. Застосовність принципів планування

Принципи планування потоків застосовні насамперед до багатопотокових систем із реалізацією схеми 1:1 (тут плануються винятково потоки ядра), а також до систем з реалізацією моделі процесів. В останньому випадку замість терміна «потік» можна вживати термін «процес», а інформацію, необхідну для планування, зберігати в структурах даних процесів. Складніші принципи планування використовують у багатопотокових системах, для яких кількість потоків користувача не збігається з кількістю потоків ядра (схеми 1:*M* і *M*:*N*). Для них потрібні два планувальники: один для роботи на рівні ядра, інший — у режимі користувача.

4.2. Види планування

Розрізняють планування довготермінове (long-term scheduling), середньотермінове (medium-term scheduling) і короткотермінове (short-term scheduling).

4.2.1. Довготермінове планування

Засоби довготермінового планування визначають, яку з програм треба завантажити у пам'ять для виконання. Таке планування називають також статичним, оскільки воно не залежить від поточного стану системи. Воно відігравало важливу роль у пакетних системах, коли заздалегідь відомо, які процеси повинні бути виконані і можна скласти розклад виконання задач. В інтерактивних системах (наприклад, у системах з розподілом часу) завантаження процесів у пам'ять здійснюють переважно користувачі, і це плануванню не підлягає; тому в них зазвичай використовують спрощену стратегію довготермінового планування. Система дає можливість створювати процеси і потоки до досягнення деякої максимально можливої межі, після чого подальші спроби створити новий процес або потік спричинятимуть помилку. Така стратегія ґрунтується і на психології користувачів, які, почуваючи себе некомфортно в перевантаженій системі, можуть переривати роботу з нею, що призводить до зниження навантаження.

4.2.2. Середньотермінове планування

Засоби середньотермінового планування керують переходом потоків із призупиненого стану в стан готовності й назад. Відразу ж зазначимо, що керуючі блоки готових до виконання потоків організуються у пам'яті в структуру, яку називають чергою готових потоків (ready queue). Докладніше розглянемо цю чергу під час вивчення короткотермінового планування.

Перехід потоку в призупинений стан можуть викликати такі фактори:

- очікування операції введення-виведення;
- очікування закінчення виконання іншого потоку (приєднання);
- блокування потоку через необхідність його синхронізації з іншими потоками.

Зазвичай для коректної організації такого очікування, крім черги готових потоків, реалізують додатковий набір черг. Кожна така черга пов'язана з ресурсом, який може викликати очікування потоку (наприклад, із пристроєм введення-виведення); ці черги ще називають *чергами планування* (scheduling queues) або *чергами очікування* (wait queues). Середньотерміновий планувальник керує всіма цими чергами, переміщаючи потоки між ними та чергою готових потоків. На рис. 4.2 зображена структура черг планування.

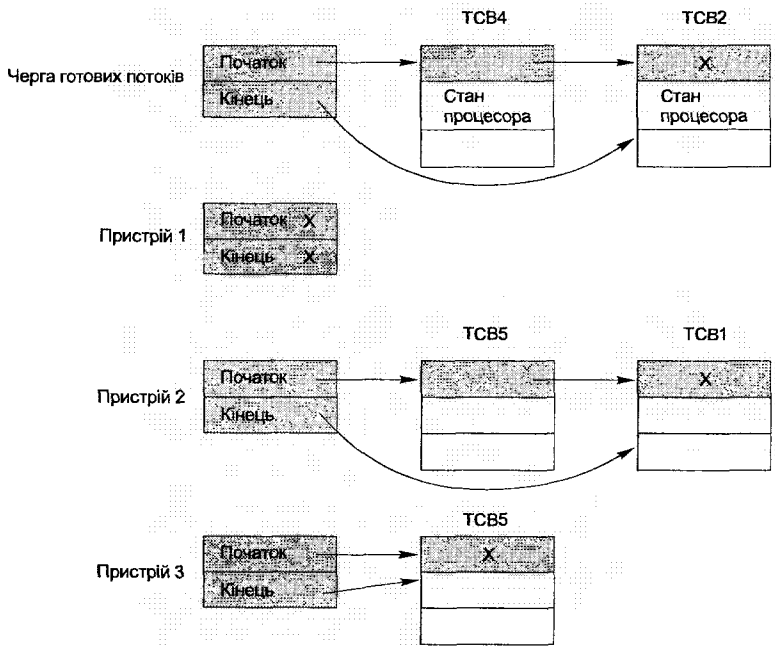

Рис. 4.2. Організація черг готових потоків і планування

Особливості планування операцій введення-виведення розглянемо в розділі 15.

4.2.3. Короткотермінове планування

Короткотермінове планування, або *планування процесора* (CPU scheduling), є найважливішим видом планування. Воно дає змогу відповісти на два базових запитання.

- Коли перервати виконання потоку?
- Якому потокові з числа готових до виконання потрібно передати процесор у цей момент?

Короткотерміновий планувальник – це підсистема ОС, яка в разі необхідності перериває активний потік і вибирає з черги готових потоків той, що має виконуватися. До його продуктивності ставлять найвищі вимоги, бо він отримує керування дуже часто. Виділяють також диспетчер (dispatcher), який безпосередньо передає керування вибраному потокові (перемикає контекст).

Формат черги готових потоків залежить від реалізації короткотермінового планування. Така черга може бути організована за принципом FIFO, бути чергою із пріоритетами, деревом або невпорядкованим зв'язним списком.

Усі стратегії й алгоритми планування, які ми будемо розглядати далі, належать до короткотермінового планування.

4.3. Стратегії планування. Витісняльна і невитісняльна багатозадачність

Перед тим як розглянути основні стратегії планування, перелічимо варіанти передачі керування від одного потоку до іншого:

- після того, як потік перейшов у стан очікування (наприклад, під час введення-виведення або приєднання);
- після закінчення виконання потоку;
- явно (потік сам віддає процесор іншим потокам на час, поки він не зайнятий корисною роботою);
- за перериванням (наприклад, переривання від таймера дає змогу перервати потік, що виконується довше, ніж йому дозволено).

Останній варіант відрізняється від інших тим, що потік не може контролювати, коли настане час передачі керування, за це відповідає планувальник операційної системи. Залежно від підтримки такого варіанта передачі керування розрізняють дві основні стратегії планування потоків – витісняльну і невитісняльну багатозадачність.

При *витісняльній багатозадачності* (preemptive multitasking) потоки, що логічно мають виконуватися, можуть бути тимчасово перервані планувальником ОС без їхньої участі для передачі керування іншим потокам. Переривання виконання потоку й передачу керування іншому потокові найчастіше здійснюють в обробнику переривання від системного таймера. Така стратегія реалізована в усіх сучасних ОС, і тому буде розглянута в цьому розділі докладніше.

При *невитісняльній багатозадачності* (non-preemptive multitasking) потоки можуть виконуватися упродовж необмеженого часу й не можуть бути перервані ОС. Для невитісняльної багатозадачності передача керування за останнім варіантом не реалізована, і потоки самі повинні віддавати керування ОС для передачі іншим потокам або, принаймні, переходити у стан очікування. Якщо якийсь потік забуде або не зможе це зробити, наприклад займе процесор нескінченним циклом, інші потоки не зможуть продовжувати свою роботу. Таку стратегію було реалізовано в ОС Novell NetWare (наприклад, у версії 3.11, яку широко використовували в 90-х роках XX ст).

Природно, що реалізація невитісняльної багатозадачності в загальному випадку робить систему досить нестабільною (будь-яке некоректно написане застосування користувача може спричинити «зависання» всієї системи). Практика показує, що невитісняльна багатозадачність у системах із застосуваннями користувача не може бути реалізована.

Таку стратегію, проте, можна використати в системах, де всі застосування виконуються в режимі ядра і фактично є системними драйверами. Для розробки таких застосувань необхідна висока кваліфікація програмістів, вимоги до надійності застосувань можна порівняти з вимогами до самої ОС. При цьому простота реалізації та відсутність зовнішніх переривань потоків від планувальника ОС може підвищити продуктивність системи для обмеженого кола задач (наприклад, у випадку ОС NetWare це було використання системи як файлового сервера).

4.4. Алгоритми планування

Як ми вже знаємо, алгоритм планування дає змогу короткотерміновому планувальникові вибирати з готових до виконання потоків той, котрий потрібно виконувати наступним. Можна сказати, що алгоритми планування реалізують політику планування.

Залежно від стратегії планування, яку реалізують алгоритми, їх поділяють на витісняльні та невитісняльні. Витісняльні алгоритми переривають потоки під час їхнього виконання, невитісняльні – не переривають. Деякі алгоритми відповідають лише одній із цих стратегій, інші можуть мати як витісняльний, так і невитісняльний варіанти реалізації.

4.4.1. Планування за принципом FIFO

Розглянемо найпростіший («наївний») невитісняльний алгоритм, у якому потоки ставлять на виконання в порядку їхньої появи у системі й виконують до переходу в стан очікування, явної передачі керування або завершення. Чергу готових потоків при цьому організовують за принципом FIFO, тому алгоритм називають алгоритмом FIFO.

Як тільки в системі створюється новий потік, його керуючий блок додається у хвіст черги. Коли процесор звільняється, його надають потоку з голови черги.

У такого алгоритму багато недоліків:

- він за визначенням є невитісняльним;
- середній час відгуку для нього може бути доволі значним (наприклад, якщо першим надійде потік із довгим інтервалом використання процесора, інші потоки чекатимуть, навіть якщо вони самі використовують тільки короткі інтервали);
- він підлягає ефекту конвою (convoy effect).

Ефект конвою можна пояснити такою ситуацією. Припустимо, що в системі є один потік (назвемо його T_{cpu}), обмежений можливостями процесора, і багато потоків T_{io}, обмежених можливостями введення-виведення. Рано чи пізно потік T_{cpu} отримає процесор у своє розпорядження і виконуватиме інструкції з довгим

інтервалом використання процесора. За цей час інші потоки T_{io} завершать введення-виведення, перемістяться в чергу готових потоків і там чекатимуть, при цьому пристрої введення-виведення простоюватимуть. Коли T_{cpu} нарешті заблокують і відбудеться передача керування, всі потоки T_{io} швидко виконають інструкції своїх інтервалів використання процесора (у них, як ми знаємо, такі інтервали короткі) і знову перейдуть до введення-виведення. Після цього T_{cpu} знову захопить процесор на тривалий час і т. д.

4.4.2. Кругове планування

Найпростішим для розуміння і найсправедливішим витісняльним алгоритмом є *алгоритм кругового планування* (round-robin scheduling). У середні віки терміном «round robin» називали петиції, де підписи йдуть по колу, щоб не можна було дізнатися, хто підписався першим (ця назва свідчить, що для такого алгоритму всі потоки рівні).

Кожному потокові виділяють інтервал часу, який називають *квантом часу* (time quantum, time slice) і упродовж якого цьому потокові дозволено виконуватися. Коли потік усе ще виконується після вичерпання кванта часу, його переривають і перемикають процесор на виконання інструкцій іншого потоку. Коли він блокується або закінчує своє виконання до вичерпання кванта часу, процесор теж передають іншому потокові. Довжина кванта часу для всієї системи однакова.

Такий алгоритм реалізувати досить легко. Для цього черга готових потоків має бути циклічним списком. Коли потік вичерпав свій квант часу, його переміщують у кінець списку, туди ж додають і нові потоки (рис. 4.3). Перевірку вичерпання кванта часу виконують в обробнику переривання від системного таймера.

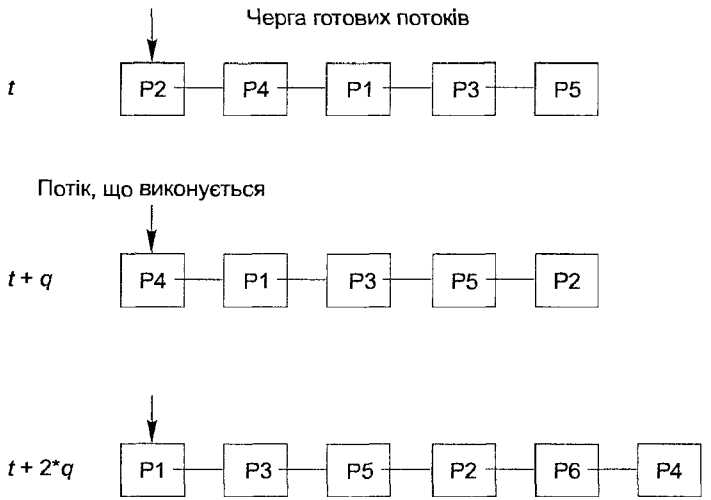

Рис. 4.3. Кругове планування

Єдиною характеристикою, яка впливає на роботу алгоритму, є довжина кванта часу. Тут слід дотримуватися балансу між часом, що витрачається на перемикання контексту, і необхідністю відповідати на багато одночасних інтерактивних запитів.

Є різні способи розв'язання проблеми голодування. Наприклад, планувальник може поступово зменшувати пріоритет потоку, який виконують (такий процес називають старінням), і коли він стане нижче, ніж у наступного за пріоритетом потоку, перемкнути контекст на цей потік. Можна, навпаки, поступово підвищувати пріоритети потоків, які очікують.

4.4.4. Планування на підставі характеристик подальшого виконання

Важливим класом алгоритмів планування з пріоритетами є алгоритми, в яких рішення про вибір потоку для виконання приймають на підставі знання або оцінки характеристик подальшого його виконання.

Насамперед слід відзначити алгоритм *«перший — із найкоротшим часом виконання»* (Shortest Time to Completion First, STCF), коли з кожним потоком пов'язують тривалість наступного інтервалу використання ним процесора і для виконання щоразу вибирають той потік, у якого цей інтервал найкоротший. У результаті потоки, що захоплюють процесор на коротший час, отримують під час планування перевагу і швидше виходять із системи.

Алгоритм STCF є теоретично оптимальним за критерієм середнього часу відгуку, тобто можна довести, що для вибраної групи потоків середній час відгуку в разі застосування цього алгоритму буде мінімальним порівняно з будь-яким іншим алгоритмом. На жаль, для короткотермінового планування реалізувати його неможливо, тому що ця реалізація потребує передбачення очікуваних характеристик (у розділі 9 буде показано, що це не останній теоретично оптимальний алгоритм із таким недоліком). Для довготермінового планування його використовують досить часто (у цьому разі, ставлячи задачу на виконання, оператор повинен вказати очікуваний момент її завершення, на який система буде зважати під час вибору). Зазначимо також, що оптимальність такого алгоритму невіддільна від його «несправедливості» до потоків із довшими інтервалами використання процесора.

Для короткотермінового планування може бути реалізоване наближення до цього алгоритму, засноване на оцінці довжини чергового інтервалу використання процесора з урахуванням попередніх інтервалів того самого потоку. Для обчислення цієї оцінки можна використати рекурсивну формулу

$$t_{n+1} = \alpha T_n + (1 - \alpha) t_n, \quad 0 \le \alpha \le 1, \quad t_0 = T_0,$$

де t_{n+1} — оцінка довжини інтервалу; t_n — оцінка довжини попереднього інтервалу; T_n — справжня довжина попереднього інтервалу. Найчастіше використовують значення $\alpha = 0{,}5$, у цьому разі для перерахування оцінки достатньо обчислити середнє між попередньою оцінкою і реальним значенням інтервалу.

Витісняльним аналогом STCF є *алгоритм «перший — із найкоротшим часом виконання, що залишився»* (Shortest Remaining Time to Completion First, SRTCF). Його відмінність від SCTF полягає в тому, що, коли в чергу готових потоків додають новий, у якого наступний інтервал використання процесора коротший, ніж час, що залишився до завершення виконання поточного потоку, поточний потік переривається, і на його місце стає новий потік.

4.4.5. Багаторівневі черги зі зворотним зв'язком

Алгоритм *багаторівневих черг зі зворотним зв'язком* (multilevel feedback queues) є найуніверсальнішим алгоритмом планування (за допомогою налаштування параметрів його можна звести майже до будь-якого іншого алгоритму), але при цьому одним із найскладніших у реалізації.

З погляду організації структур даних цей алгоритм схожий на звичайний алгоритм багаторівневих черг: є кілька черг готових потоків із різним пріоритетом, при цьому потоки черги із нижчим пріоритетом виконуються, тільки коли всі черги верхнього рівня порожні.

Відмінності між двома алгоритмами полягають у тому, що:

- потокам дозволено переходити з рівня на рівень (із черги в чергу);
- потоки в одній черзі об'єднуються не за пріоритетом, а за довжиною інтервалу використання процесора, потоки із коротшим інтервалом перебувають у черзі з більшим пріоритетом.

Усередині всіх черг, крім найнижчої, використовують кругове планування (у найнижчій працює FIFO-алгоритм). Різні черги відповідають різній довжині кванта часу — що вищий пріоритет, то коротший квант (звичайно довжина кванта для сусідніх черг зменшується удвічі). Якщо потік вичерпав свій квант часу, він переміщається у хвіст черги із нижчим пріоритетом (і з довшим квантом). У результаті потоки з коротшими інтервалами (наприклад, обмежені введенням-виведенням) залишаються з високим пріоритетом, а потоки з довшими інтервалами подовжують свій квант часу (рис. 4.4). Можна також автоматично переміщати потоки, які давно не отримували керування, із черги нижнього рівня на рівень вище.

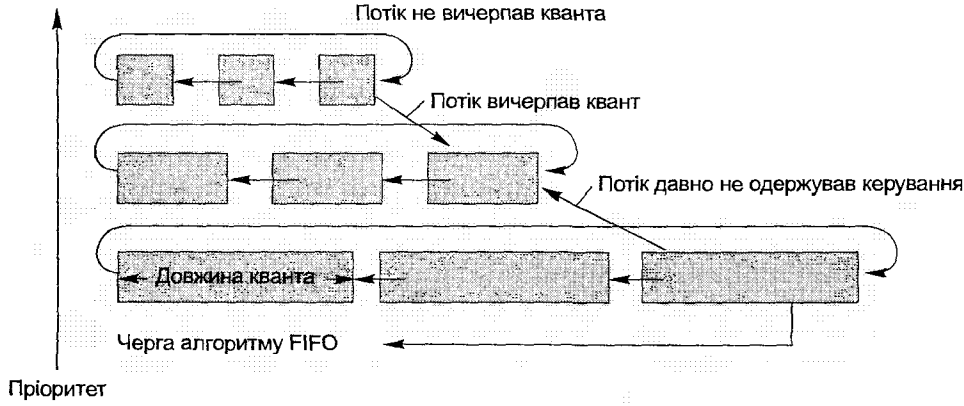

Рис. 4.4. Багаторівневі черги зі зворотним зв'язком

4.4.6. Лотерейне планування

Лотерейне планування (lottery scheduling) — це останній алгоритм, який ми розглянемо. Він простий для розуміння і легкий у реалізації, разом з тим має великі можливості.

Ідея лотерейного планування полягає у тому, що:

- потік отримує деяку кількість лотерейних квитків, кожен з яких дає право користуватися процесором упродовж часу T;
- планувальник через проміжок часу T вибирає випадково один лотерейний квиток;
- потік, «що виграв», дістає керування до наступного розіграшу.

Лотерейне планування дає змогу:

- емулювати кругове планування, видавши кожному потокові однакову кількість квитків;
- емулювати планування із пріоритетами, розподіляючи квитки відповідно до пріоритетів потоків;
- емулювати SRTCF – давати коротким потокам більше квитків, ніж довгим (якщо потік отримав хоча б один квиток, він не голодуватиме);
- забезпечити розподіл процесорного часу між потоками – дати кожному потокові кількість квитків, пропорційну до частки процесорного часу, який потрібно йому виділити (наприклад, якщо є три потоки, і треба, щоб потік A займав 50 % процесора, а потоки B і C – по 25 %, можна дати потоку A два квитки, а потокам B і C – по одному);
- динамічно міняти пріоритети, відбираючи і додаючи квитки по ходу.

Хоча більшість цих задач може бути розв'язана лотерейним плануванням лише приблизно, з деякою ймовірністю, на практиці отримують цілком задовільні результати. При цьому що довше працює система, то ближчими будуть результати до теоретичних значень (за законом великих чисел). Насправді лотерейне планування використовує той факт, що вся ідеологія планування значною мірою є евристичною, оскільки не можна точно передбачити характер поведінки потоків у системі.

4.5. Реалізація планування в Linux

У цьому розділі розглянемо два варіанти реалізації планування в Linux – традиційну (належить до ядер версій до 2.4 включно) і нову, включену в ядро версії 2.6.

Ядро Linux при плануванні не розрізняє процеси і потоки, тому для визначеності ми надалі говоритимемо про планування процесів.

Усі процеси в системі можна поділити на три групи: реального часу із плануванням за принципом FIFO, реального часу із круговим плануванням, звичайні.

4.5.1. Планування процесів реального часу в ядрі

Стосовно процесів реального часу, достатньо сказати, що:

- вони завжди матимуть під час планування пріоритет перед звичайними процесами;
- процес із плануванням за принципом FIFO виконують доти, поки він сам не віддасть процесор (наприклад, внаслідок призупинення або завершення) або поки не буде витиснений процесом реального часу із вищим пріоритетом;
- те саме стосується процесу із круговим плануванням, крім того, що він додатково буде витіснений після вичерпання кванта часу.

4.5.2. Традиційний алгоритм планування

Розглянемо алгоритм планування звичайних процесів [62]. В основі алгоритму лежить розподіл процесорного часу на *епохи* (epochs). Упродовж епохи кожен процес має квант часу, довжину якого розраховують у момент початку епохи. Здебільшого різні процеси мають кванти різної довжини. Коли процес вичерпав свій квант, його витісняють і протягом поточної епохи він більше не виконуватиметься. Керування передають іншому процесові. Якщо ж процес був призупинений для виконання введення-виведення або внаслідок синхронізації, його квант не вважають вичерпаним і він може бути вибраний планувальником упродовж поточної епохи. Епоха закінчується, коли всі готові до виконання процеси вичерпали свої кванти. У цьому разі алгоритм планування перераховує кванти для всіх процесів і розпочинає нову епоху.

Квант, який задають на початку епохи, називають *базовим квантом часу процесу*. Його значення можуть динамічно змінюватися системними викликами `nice()` і `setpriority()`. Процес-нащадок завжди успадковує базовий квант свого предка.

Пріоритет процесу буває двох видів: фіксований, для процесів реального часу, що задають тільки під час створення процесу, та динамічний, для звичайних процесів, який залежить від базового пріоритету і часу, що залишився до вичерпання кванта. Динамічний пріоритет будь-якого звичайного процесу завжди нижчий за будь-який пріоритет процесу реального часу.

Опишемо найважливіші поля структури даних процесу стосовно планування:

- `policy` – визначає, до якої групи відноситься процес (звичайні, реального часу з алгоритмом FIFO тощо);
- `nice` – задає величину, на якій ґрунтується базовий квант часу процесу (надалі для спрощення вважатимемо `nice` рівним базовому кванту, насправді це не зовсім так);
- `counter` – містить кількість переривань таймера, що залишилися до вичерпання кванта часу процесу. На початку епохи `counter` надають значення базового кванта і зменшують його на одиницю в обробнику переривання таймера.

Умови виклику процедури планування

Розглянемо ситуації, коли відбувається виклик процедури планування (її називають `schedule()`).

- Коли процес повинен бути заблокований через те, що потрібний йому ресурс у цей час недоступний. У цьому разі його керуючий блок спочатку додають у відповідну чергу очікування, а потім відбувається перепланування.
- За допомогою *відкладеного запуску* (lazy invocation). Відкладений запуск полягає в тому, що у певний момент часу спеціальному полю `need_resched` структури процесу надають значення 1. Це відбувається в таких випадках: коли поточний процес вичерпав свій квант; коли у стан готовності переходить процес, пріоритет якого вищий, ніж у поточного; коли процес явно поступається своїм правом виконання через відповідний системний виклик. При цьому негайного перепланування не відбувається, але пізніше, коли цей процес повинен знову отримати керування після переривання, він перевіряє, чи не дорівнює поле `need_resched` одиниці. Якщо рівність виконується, запускають процедуру планування.

Процедура планування

Ця процедура спочатку перевіряє, чи не переходить поточний процес у стан очікування, і якщо це так, вилучає його з черги готових процесів. Потім вибирається процес для виконання. Для цього проглядають чергу готових процесів, для кожного процесу оцінюють динамічний пріоритет і вибирають процес із максимальним його значенням. Алгоритм оцінки цього пріоритету описаний нижче. Для процесу, що вичерпав свій квант часу, він дорівнюватиме нулю.

Якщо жоден процес не був вибраний, поточний процес продовжує виконуватися. Коли ж вибір відбувся, контекст перемикають на новий процес.

Початок нової епохи

Особлива ситуація виникає тоді, коли для всіх процесів у черзі готових процесів значення динамічного пріоритету дорівнює нулю, тобто всі вони вичерпали свій квант і настав час починати нову епоху. Проте це не означає, що система взагалі не має процесів, для яких квант не вичерпаний, — вони можуть перебувати в чергах очікування (найчастіше це процеси, обмежені введенням-виведенням).

Коли розпочинається нова епоха, відбувається перерахування квантів для всіх процесів системи (не тільки для процесів у стані готовності). При цьому довжину кванта для кожного процесу задають рівною сумі його базового пріоритету і половини частини кванта, що залишилася в нього:

```
for_each_task (p)
    p.counter = (p.counter / 2) + p.nice;
```

Оскільки до початку нової епохи ненульовий квант залишається тільки у процесів, які не перебувають у стані готовності, цей алгоритм надає певну перевагу процесам, обмеженим можливостями введення-виведення. При цьому значення кванта для процесу ніколи не зможе стати більшим, ніж подвоєне значення його базового пріоритету.

Розрахунок динамічного пріоритету

Тепер повернемося до обчислення динамічного пріоритету процесу. Для цього використовують функцію `goodness()`. Розглянемо можливі значення, які вона може повернути.

- 0 — у разі, коли процес вичерпав свій квант часу. Цей процес не буде вибраний для виконання, крім випадку, коли він стоїть у черзі готових процесів першим, а всі інші процеси черги також вичерпали свій квант.
- Від 0 до 1000 — у разі, коли процес не вичерпав свого кванту часу. Це значення розраховують на основі значення базового кванта процесу й частини поточного кванта, що залишилася в нього. Спрощено це можна зобразити так:

  ```
  c = p.counter + p.nice;
  ```

  де `p` — покажчик на керуючий блок процесу.

Звідси випливає, що більше часу залишилося процесу для виконання і що довший його базовий квант, то вищий його пріоритет. Крім того, це значення додатково збільшують на одиницю для процесів, які використовують ту саму пам'ять, що й предки (наприклад, якщо процес відображає потік, створений за допомогою функції `clone()`).

Перерахування кванта під час створення нового процесу

Тепер розглянемо, що відбувається під час створення нового процесу. Найпростіше рішення (копіювати значення counter у структуру даних нащадка) може призвести до того, що процеси будуть штучно подовжувати свій квант створенням нових нащадків, виконуючих той самий код. Для того щоб цьому перешкодити, після функції fork() значення counter розділяють навпіл: одна половина переходить нащадкові, інша залишається предкові.

Перелічимо недоліки алгоритму.

- Вибір процесу для виконання відбувається внаслідок розрахунку динамічного пріоритету для всіх процесів у черзі готових процесів. Зі збільшенням кількості готових процесів у системі переглядати цю чергу від початку до кінця під час кожного виклику процедури планування стає невигідно.
- Якщо кількість процесів буде дуже великою, перерахування всіх динамічних пріоритетів на початку нової епохи може виконуватися досить довго. З іншого боку, епохи змінюються рідше, що більше в системі процесів.
- Алгоритм розрахований на зменшення часу відгуку для процесів, обмежених можливостями введення-виведення, навіть якщо вони не є інтерактивними (наприклад, фоновий процес індексації пошукової системи) і не потребують малого часу відгуку.
- Зі збільшенням кількості процесорів підтримувати загальні черги, які не враховують наявність різних процесорів, стає невигідно.

4.5.3. Сучасні підходи до реалізації планування

Зазначені недоліки починали істотно впливати на роботу системи, коли вона функціонувала за умов граничного навантаження. У звичайних умовах традиційне планування в Linux працювало досить ефективно.

Проте робота над виправленням недоліків тривала. Як наслідок, у ядро версії 2.6 була інтегрована нова реалізація алгоритму планування [97]. Розглянемо коротко, як вона допомагає розв'язувати названі раніше проблеми.

Насамперед, цей алгоритм підтримує окремі черги готових процесів для кожного процесора, забезпечуючи ефективну роботу за умов багатопроцесорності.

Ще одна проблема, яку має розв'язати новий алгоритм, пов'язана з необхідністю розраховувати у старому алгоритмі динамічний пріоритет для всіх готових процесів під час кожного виклику процедури планування. Рішення приймають таке: кожна черга готових процесів — це масив черг готових процесів, де елементи упорядковані за динамічним пріоритетом. У результаті під час вибору процесу для виконання достатньо продивитись чергу з найвищим пріоритетом до першого процесу, який можна запустити. Ця процедура не залежить від загальної кількості готових процесів у системі.

Є два масиви черг готових процесів — масив черг активних процесів і масив черг процесів з вичерпаним квантом. Після того як процес вичерпав свій квант, його переносять із першого масиву в другий. Коли в масиві активних черг не залишається жодного процесу, обидва масиви міняються місцями, і послідовність кроків повторюють із самого початку. У підсумку з вичерпанням квантів процесами підвищується ймовірність запуску тих процесів, які до цього часу ще не одержували керування.

4.5.4. Програмний інтерфейс планування

У цьому розділі розглянемо системні виклики Linux, за допомогою яких можна працювати із базовим пріоритетом процесів (величиною nice) і цим впливати на їхнє планування.

Для зміни базового пріоритету процесу використовують виклик setpriority():

```
#include <sys/resource.h>
int setpriority(int which, int who, int priority);
```

Зокрема, параметр which може набувати значення PRIO_PROCESS або PRIO_USER, відповідно показуючи, що параметр who буде інтерпретований як ідентифікатор процесу чи ідентифікатор користувача. У першому випадку задають пріоритет для конкретного процесу (або для поточного процесу, якщо who дорівнює нулю), у другому — для всіх процесів цього користувача.

Параметр priority задає новий пріоритет. Пріоритет може варіюватися в межах від –20 до 20, менші значення свідчать про вищий пріоритет. Значенням за замовчуванням є 0. Негативні значення priority можуть задавати лише користувачі з правами адміністратора.

Для отримання інформації про поточний базовий пріоритет використовують виклик getpriority():

```
int getpriority(int which, int who);
```

Цей виклик повертає значення пріоритету, параметри which і who для нього мають той самий зміст, що й для функції setpriority().

Розглянемо приклад використання цих викликів:

```
// задати пріоритет для поточного процесу
setpriority(PRIO_PROCESS, 0, 10);
// довідатися про поточне значення пріоритету
printf ("поточний пріоритет: %d\n", getpriority(PRIO_PROCESS, 0));
```

Для відносної зміни базового пріоритету поточного процесу можна також використати системний виклик nice():

```
#include <unistd.h>
int nice(int inc);   // змінює пріоритет поточного процесу на inc
```

4.6. Реалізація планування у Windows XP

4.6.1. Планування потоків у ядрі

Ядро Windows XP розв'язує під час планування дві основні задачі [14, 97]:

- облік відносних пріоритетів, присвоєних кожному потокові;
- мінімізацію часу відгуку інтерактивних застосувань.

Базовою одиницею планування є потік. Під час планування ядро не розрізняє потоки різних процесів, воно має справу з пріоритетами потоків, готових до виконання в певний момент часу.

Під час планування ядро працює з мінімальними версіями потоків (блоками KTHREAD). У них зберігається така інформація, як загальний час виконання потоку, його базовий і поточний пріоритет, диспетчерський стан потоку (готовність, очікування, виконання тощо).

Пріоритети потоків і процесів

Для визначення порядку виконання потоків диспетчер ядра використовує систему пріоритетів. Кожному потокові присвоюють пріоритет, заданий числом у діапазоні від 1 до 31 (що більше число, то вище пріоритет). Пріоритети реального часу – 16–31; їх резервує система для дій, час виконання яких є критичним чинником. Динамічні пріоритети – 1–15; вони можуть бути присвоєні потокам застосувань користувача.

Ядро системи може надати потоку будь-який динамічний пріоритет. Win32 API не дає можливості зробити це з цілковитою точністю, у ньому використовують дворівневу систему, яка зачіпає як процес, так і його потоки: спочатку процесу присвоюють *клас пріоритету*, а потім потокам цього процесу – відносний пріоритет, який відраховують від класу пріоритету процесу (називаного ще базовим пріоритетом). Під час виконання відносний пріоритет може змінюватися.

Розрізняють такі класи пріоритету процесів: реального часу (real-time, приблизно відповідає пріоритету потоку 24); високий (high, 13); нормальний (normal, 8); невикористовуваний (idle, 4).

Відносні пріоритети потоку бувають такі: найвищий (+2 до базового); вище за нормальний (+1 до базового); нормальний (дорівнює базовому); нижче за нормальний (–1 від базового); найнижчий (–2 від базового).

Є два додаткових модифікатори відносного пріоритету: критичний за часом (time-critical) і невикористовуваний (idle). Перший модифікатор тимчасово задає для потоку пріоритет 15 (найвищий динамічний пріоритет), другий аналогічним чином задає пріоритет 1.

Особливості задання кванта часу

Важливою характеристикою системи є довжина кванта часу. Розрізняють короткі й довгі кванти, для яких можна задати змінну та фіксовану довжину.

У Windows XP інтерактивно можна задавати таку довжину кванта (вибирають Settings (Параметры) у групі Performance (Быстродействие) на вкладці Advanced (Дополнительно) вікна властивостей My Computer (Свойства системы)):

- короткі кванти змінної довжини (вкладка Advanced (Дополнительно), перемикач Programs (Программ) у групі властивостей Processor Scheduling (Распределение времени процесора)). Можлива довжина кванта – 10 або 30 мс, при цьому застосування, з яким починає працювати користувач, автоматично переходить до використання довших квантів. Ця установка надає перевагу інтерактивним процесам;
- довгі кванти фіксованої довжини (вкладка Advanced (Дополнительно), перемикач Background services (Служб, работающих в фоновом режиме) у групі властивостей Processor Scheduling (Распределение времени процесора)). Довжина кванта фіксована й дорівнює 120 мс. Ця установка надає перевагу фоновим процесам.

Пошук потоку для виконання

Для виконання новий потік вибирається, коли:

- минув квант часу для потоку (з використанням *алгоритму пошуку готового потоку*);
- потік перейшов у стан очікування події (потік сам віддає квант часу і дає команду планувальникові запустити алгоритм пошуку готового потоку);

- потік перейшов у стан готовності до виконання (використовують *алгоритм розміщення готового потоку*).

Планувальник підтримує спеціальну структуру даних — *список готових потоків* (dispatcher ready list). У цьому списку зберігається 31 елемент — по одному для кожного рівня пріоритету. З кожним елементом пов'язана черга готових потоків, всі потоки з однаковим пріоритетом перебувають у черзі, яка відповідає їхньому рівню пріоритету.

Під час виконання алгоритму пошуку готового потоку планувальник переглядає всі черги потоків, починаючи з черги найвищого пріоритету (31). Як тільки під час цього перегляду трапляється потік, його відразу вибирають для виконання. За допомогою цього алгоритму вибирають перший потік непустої черги з найвищим пріоритетом. Можна сказати, що в межах однієї черги використовують алгоритм кругового планування, якщо не враховувати динамічну корекцію пріоритетів, яку ми розглянемо далі.

Алгоритм розміщення готового потоку поміщає потік у список готових потоків. Спочатку перевіряють, чи не володіє потік вищим пріоритетом, ніж той, котрий виконується в цей момент. При цьому новий потік негайно починає виконуватися, а поточний поміщається у список готових потоків; у противному разі новий потік поміщається в чергу списку готових потоків, відповідну до його пріоритету. У початку кожної черги розташовані потоки, які були витиснені до того, як вони виконувалися впродовж хоча б одного кванта, всі інші потоки поміщаються в кінець черги.

Якщо подивитися на ситуацію з боку потоку, що виконується, то важливо знати, коли він може бути витиснений. Це трапляється коли:

- потік перейшов у стан очікування;
- минув квант часу потоку;
- потік із вищим пріоритетом перейшов у стан готовності до виконання;
- змінився пріоритет потоку або пріоритет іншого потоку.

Динамічна зміна пріоритету і кванта часу

Під час виконання потоків динамічний пріоритет і довжина кванта часу можуть бути скориговані ядром системи. Розрізняють два види такої динамічної зміни: *підтримка* (boosting) і *ослаблення* (decay).

Підтримка зводиться зазвичай до тимчасового підвищення пріоритету потоків. Коли потік переходить у стан готовності до виконання внаслідок настання події, на яку він очікував, виконують операцію підтримки.

- Під час завершення операції введення-виведення підвищення пріоритету залежить від типу операції. Наприклад, після виконання дискових операцій пріоритет збільшують на одиницю, після введення із клавіатури або обробки події від миші — на 6.
- Під час зміни стану синхронізаційного об'єкта (докладніше такі об'єкти будуть розглянуті пізніше) пріоритет потоку, який очікує цієї зміни, збільшують на одиницю.
- Вихід з будь-якого стану очікування для потоків інтерактивних застосувань призводить до підвищення пріоритету на 2, таке саме підвищення відбувається

під час переходу в стан готовності потоків, пов'язаних із відображенням інтерфейсу користувача.

- Для запобігання голодуванню потоки, які не виконувалися упродовж досить тривалого часу, різко підвищують свій пріоритет (цей випадок розглянемо окремо).

Зазначимо, що внаслідок операцій підтримки динамічний пріоритет потоку не може перевищити значення 15 (максимально допустимого динамічного пріоритету). Якщо операція підтримки вимагає підвищення пріоритету до величини, вищої за це значення, пріоритет збільшують тільки до 15.

Підвищення пріоритету внаслідок підтримки дедалі слабшає. Після закінчення кожного кванта часу поточний пріоритет потоку зменшують на одиницю, поки він не дійде до базового, після чого пріоритет залишають на одному рівні до наступної операції підтримки.

Ще одним видом підтримки є зміна кванта часу для інтерактивних застосувань. Якщо під час налаштування системи задано використання квантів змінної довжини, можна вказати, що для інтерактивних застосувань довжина кванта буде збільшуватися (це називають підтримкою кванта для інтерактивних застосувань). Якщо така підтримка задана, то коли інтерактивне застосування захоплює фокус, всі його потоки отримують квант часу, який дорівнює значенню підтримки (дозволене одне з можливих значень кванта, наприклад, 40 або 60 мс).

З іншого боку, значення кванта може й зменшуватися (слабшати). Так, під час виконання будь-якої функції очікування довжина кванта зменшується на одиницю.

Для потоків із пріоритетом реального часу динамічна зміна пріоритету або довжини кванта ніколи не відбувається. Єдиний спосіб змінити пріоритет таких потоків — викликати відповідну функцію із прикладної програми.

Запобігання голодуванню

Якщо в системі постійно є потоки з високим пріоритетом, може виникати голодування потоків, пріоритет яких нижчий. Для того щоб уникнути голодування, спеціальний потік ядра один раз за секунду обходить чергу готових потоків у пошуках тих, які перебували у стані готовності досить довго (понад 3 с) і жодного разу не отримали шансу на виконання. Коли такий потік знайдено, то йому присвоюють пріоритет 15 (і він дістає змогу негайного виконання); крім того, довжину його кванта часу подвоюють. Після того, як два кванти часу минають, пріоритет потоку і його квант повертаються до вихідних значень.

Цей алгоритм не враховує причин голодування і не розрізняє потоків інтерактивних і фонових процесів.

4.6.2. Програмний інтерфейс планування

У цьому розділі розглянемо функції Win32 API, за допомогою яких можна працювати із класами пріоритетів процесів і відносних пріоритетів потоків [31].

Для зміни класу пріоритету процесу використовують функцію `SetPriorityClass()`, для визначення поточного класу пріоритету — функцію `GetPriorityClass()`:

```
BOOL SetPriorityClass(HANDLE ph, DWORD pclass);
DWORD GetPriorityClass(HANDLE ph);
```

Параметр ph визначає дескриптор процесу, для якого задають клас пріоритету, а параметр pclass — клас пріоритету. Можливі такі значення pclass:

- REALTIME_PRIORITY_CLASS (реального часу);
- HIGH_PRIORITY_CLASS (високий);
- NORMAL_PRIORITY_CLASS (нормальний);
- IDLE_PRIORITY_CLASS (невикористовуваний).

Розглянемо приклад використання цих функцій:

```
HANDLE curh = GetCurrentProcess();
// задати клас пріоритету для поточного процесу
SetPriorityClass(curh, IDLE_PRIORITY_CLASS);
// взнати поточне значення класу пріоритету
printf("Поточний клас пріоритету: %d\n", GetPriorityClass(curh));
```

Щоб задати відносний пріоритет потоку, використовують функцію SetThreadPriority(), а щоб задати його значення — GetThreadPriority().

```
BOOL SetThreadPriority(HANDLE th, int priority);
int GetThreadPriority(HANDLE th);
```

Параметр th є дескриптором потоку, параметр priority (і повернуте GetThreadPriority()) може набувати одного з таких значень:

- THREAD_PRIORITY_TIME_CRITICAL (критичний за часом);
- THREAD_PRIORITY_HIGHEST (найвищий);
- THREAD_PRIORITY_ABOVE_NORMAL (вищий за нормальний);
- THREAD_PRIORITY_NORMAL (нормальний);
- THREAD_PRIORITY_BELOW_NORMAL (нижчий за нормальний);
- THREAD_PRIORITY_LOWEST (найнижчий);
- THREAD_PRIORITY_IDLE (невикористовуваний).

Розглянемо приклад використання цих функцій.

```
DWORD tid;
// створення потоку
HANDLE th = _beginthreadex(... CREATE_SUSPENDED, &tid);
// задання пріоритету
SetThreadPriority(th, THREAD_PRIORITY_IDLE);
// поновлення виконання потоку
ResumeThread(th);
// визначення пріоритету
printf("Поточний пріоритет потоку: %d\n", GetThreadPriority(th));
// закриття дескриптора потоку
CloseHandle(th);
```

У даному прикладі ми створюємо потік у призупиненому стані, задаємо його пріоритет, а потім поновлюємо його виконання. Для цього використана функція ResumeThread(), що параметром приймає дескриптор потоку.

Висновки

- Задача планування потоків зводиться до організації виконання кількох потоків на одному процесорі так, аби у користувачів виникало враження, що вони виконуються одночасно. Цілями планування є: мінімізація часу відгуку, максимізація пропускної здатності та справедливість. До основних стратегій планування належать витісняльна й невитісняльна багатозадачність. У сучасних ОС застосовують витісняльну багатозадачність, коли рішення про перемикання контексту потоку приймають у коді ядра системи, а не в коді потоку.
- Розрізняють довготермінове, середньотермінове й короткотермінове планування. Найважливіший тут короткотерміновий планувальник, котрий використовують для прийняття рішення про те, який потік запустити на виконання в певний момент. До основних алгоритмів короткотермінового планування належать планування кругове і з пріоритетами.

Контрольні запитання та завдання

1. Процес виконується в нескінченному циклі. За допомогою якого апаратного пристрою операційна система може перехопити керування в цього процесу? Опишіть особливості використання такого пристрою.
2. Наведіть приклади програм, під час виконання яких не буде жодних проблем в ОС із витісняльною багатозадачністю, але виникнуть блокування системи в ОС із невитісняльною багатозадачністю.
3. Задано послідовність потоків (табл. 4.1):

Таблиця 4.1. Параметри послідовності потоків

Потік	Тривалість	Час надходження	Пріоритет
1	3	0	1
2	5	1	2
3	2	3	3
4	2	9	4

В комірках табл. 4.2 відобразіть, який із потоків буде займати процесор у фіксований момент часу в разі використання алгоритмів планування FIFO, RR (кругового), STCF, SRTCF і PS (з пріоритетами).

Таблиця 4.2. Завантаження процесора для різних алгоритмів планування

Час	FIFO	RR	STCF	SRTCF	PS
0					
1					
2					
...					

Пріоритет 4 є найвищим.

4. Для даних попередньої задачі обчисліть час відгуку і час до завершення кожного потоку в разі використання всіх алгоритмів планування. Обчисліть також середній час відгуку і поясніть одержані результати.
5. Як зміниться завантаження пристроїв введення-виведення із зменшенням довжини кванта у разі кругового планування? Чи збільшиться в цьому випадку час виконання окремих задач?
6. Порівняйте продуктивності алгоритмів багаторівневих черг зі зворотним зв'язком і SRTCF для потоків, обмежених введенням-виведенням.
7. Опишіть, яким чином багаторівневі черги зі зворотним зв'язком відокремлюють потоки, обмежені процесором, від потоків, обмежених введенням-виведенням.
8. Припустімо, що в системі з лотерейним плануванням кількість квитків, одержуваних потоком, прямо пропорційна частці процесорного часу, який повинен бути йому наданий:
   а) запропонуйте критерій вибору наступного потоку для виконання, що базується на частці процесорного часу, використаній потоком, і кількості квитків, що у нього залишилися;
   б) скільки квитків треба надати потоку при його створенні?
9. Деякі алгоритми планування при певних значеннях параметрів можуть емулювати один одного. Оцініть можливість прямої і зворотної емуляції таких пар алгоритмів:
   а) STCF і планування з пріоритетами;
   б) FIFO і планування з пріоритетами;
   в) FIFO і багаторівневих черг зі зворотним зв'язком;
   г) STCF і кругового планування.
10. Проаналізуйте, як будуть поводитися застосування, розроблені для розв'язання завдання 10 з розділу 3, якщо в їхньому коді змінювати пріоритети потоків T і t.

Розділ 5

Взаємодія потоків

- Принципи та проблеми взаємодії потоків
- Задачі синхронізації
- Механізми синхронізації: семафори, м'ютекси та умовні змінні
- Взаємодія потоків у Linux
- Взаємодія потоків у Windows XP

У цьому розділі розглянемо основні принципи взаємодії потоків одного процесу. Основну увагу буде зосереджено на синхронізації доступу до спільно використовуваних даних таких потоків.

5.1. Основні принципи взаємодії потоків

Потоки, які виконуються в рамках процесу паралельно, можуть бути незалежними або взаємодіяти між собою.

Потік є незалежним, якщо він не впливає на виконання інших потоків процесу, не зазнає впливу з їхнього боку, та не має з ними жодних спільних даних. Його виконання однозначно залежить від вхідних даних і називається детермінованим.

Усі інші потоки є такими, що взаємодіють. Ці потоки мають дані, спільні з іншими потоками (вони перебувають в адресному просторі їхнього процесу). Їх виконання залежить не тільки від вхідних даних, але й від виконання інших потоків, тобто вони є *недетермінованими* (далі розглянемо докладно приклади такої недетермінованості).

Результати виконання незалежного потоку завжди можна повторити, чого не можна сказати про потоки, що взаємодіють.

Дані, які є загальними для кількох потоків, називають спільно використовуваними даними (shared data). Це — найважливіша концепція багатопотокового програмування. Усякий потік може в будь-який момент часу змінити такі дані. Механізми забезпечення коректного доступу до спільно використовуваних даних називають механізмами синхронізації потоків.

Працювати із незалежними потоками простіше, ніж із тими, що взаємодіють. Програміст може не враховувати того, що одночасно з таким потоком виконуються інші, а також не звертати уваги на стан спільно використовуваних даних, з якими працює потік.

Проте обійтися без реалізації взаємодії потоків неможливо з кількох причин.

- Необхідно організовувати спільне використання інформації під час роботи з потоками. Наприклад, користувачі бази даних або веб-сервера можуть захотіти одночасно виконати запити на отримання однієї й тієї самої інформації, і система має забезпечити її паралельне отримання потоками, що обслуговують цих користувачів.
- Коректна реалізація такої взаємодії та використання відповідних алгоритмів можуть значно прискорити обчислювальний процес на багатопроцесорних системах. При цьому задачі розділяють на підзадачі, які виконують паралельно на різних процесорах, а потім їхні результати збирають разом для отримання остаточного розв'язання. Таку технологію називають *технологією паралельних обчислень*.
- У задачах, що вимагають паралельного виконання обчислень та операцій введення-виведення, потоки, що виконують введення-виведення, повинні мати можливість подавати сигнали іншим потокам із завершенням своїх операцій.
- Подібна організація дає змогу розбивати задачі на окремі виконувані модулі, оформлені як окремі потоки, при цьому вихід одного модуля може бути входом для іншого, а також підвищується гнучкість системи, оскільки окремі модулі можна міняти, не чіпаючи інших.

Необхідність організації паралельного виконання потоків, що взаємодіють, потребує наявності механізмів обміну даними між ними і забезпечення їхньої синхронізації.

5.2. Основні проблеми взаємодії потоків

5.2.1. Проблема змагання

Розглянемо на найпростішому прикладі, що може відбутися, коли потоки, які взаємодіють, разом використовуватимуть спільні дані без додаткових заходів із забезпечення синхронізації.

Уявімо, що при банківської організації системи для обслуговування кожного користувача виділяють окремий потік (чим намагаються підвищити продуктивність системи у разі великої кількості одночасних запитів). Припустимо, що поміщення даних на вклад користувача зводиться до збільшення глобальної змінної `total_amount`. У цьому разі кожен потік під час зміни вкладу виконує такий найпростіший оператор:

```
total_amount = total_amount + new_amount;
```

Виникає запитання: чи можна дати гарантію, що внаслідок роботи із вкладом потік, який відповідає кожному користувачу, буде здатний збільшити значення `total_amount` на потрібну величину?

Насправді цей на перший погляд найпростіший оператор зводиться до послідовності таких дій:

- отримання поточного значення `total_amount` із глобальної пам'яті;
- збільшення його на `new_amount` і збереження результату в глобальній пам'яті.

Розглянемо випадок, коли два користувачі A і B спільно користуються одним і тим самим рахунком. На рахунку є 100 грошових одиниць. Користувач A збирається покласти на рахунок 1000, користувач B у цей самий час — 100. Потік T_A відповідає користувачу A, потік T_B — користувачу B. Розглянемо таку послідовність подій (варіант 1).

1. Потік T_A зчитує значення `total_amount`, рівне 100.
2. Потік T_B зчитує значення `total_amount`, теж рівне 100.
3. Потік T_A збільшує зчитане на кроці 1 значення `total_amount` на 1000, отримує 1100 і зберігає його у глобальній пам'яті.
4. Потік T_B збільшує зчитане на кроці 2 значення `total_amount` на 100, отримує 200 і теж зберігає його у глобальній пам'яті, перезаписуючи те, що зберіг потік T_A.

У результаті внесок користувача A повністю втрачено.

Тепер розглянемо іншу послідовність подій (варіант 2).

1. Потік T_A зчитує `total_amount`, збільшує його на 1000 і записує значення 1100 у глобальну пам'ять.
2. Потік T_B зчитує `total_amount`, рівний 1100, збільшує його на 100 і записує значення 1200 у глобальну пам'ять.

У результаті обидва внески зареєстровані успішно.

Як бачимо, результат виконання наведеного раніше найпростішого фрагмента коду залежить від послідовності виконання потоків у системі. Це спричиняє до такого: в одній ситуації код може працювати, в іншій — ні, і передбачити появу помилки в загальному випадку неможливо. Таку ситуацію називають *станом гонок* або *змаганням* (race condition), що є однією з найбільш важко вловлюваних помилок, з якими зіштовхуються програмісти. Вона практично не піддається традиційному налагодженню (оскільки нереально перебрати у налагоджувачі всі можливі комбінації послідовностей виконання потоків, особливо якщо їх багато).

Спроби розв'язувати подібні проблеми викликали необхідність *синхронізації потоків*. Відразу ж зазначимо, що проблеми синхронізації й організації паралельних обчислень є одними з найскладніших у практичному програмуванні. Тому розробку й особливо налагодження багатопотокових програм часто сприймають як своєрідне «мистецтво», що доступне далеко не всім програмістам.

Насправді така розробка та налагодження — це аж ніяк не мистецтво, а строга дисципліна, що підлягає одному головному принципу: оскільки для багатопотокових програм традиційне налагодження не придатне, програміст має писати код таким чином, щоб уже на етапі розробки не залишити місця для помилок синхронізації. У цьому розділі ознайомимося із правилами, яких треба дотримуватися, аби створений код відповідав цьому принципу.

Розглянемо основні підходи до розв'язання проблеми змагань.

Іноді (але досить рідко) можна просто ігнорувати такі помилки. Це може мати сенс, коли нас цікавить не точна реєстрація тих або інших даних, а збір статистики про них, тому окремі помилки не позначатимуться на загальному результаті. Наприклад, глобальним лічильником є величина, на базі якої розраховують середню кількість запитів до системи за добу і можна проігнорувати помилки реєстрації таких запитів, що трапляються раз на кілька годин. На жаль, у більшості випадків такий підхід не прийнятний.

Іноді використання глобальних даних не диктується специфікою задачі. У цьому разі однозначним розв'язанням є створення локальних копій цих даних для кожного потоку й оперування тільки ними. На практиці це працює вельми добре й має використовуватися, де тільки можливо. Наприклад, якщо специфіка задачі допускає створення окремого лічильника для кожного потоку (або глобального масиву лічильників, де кожний елемент змінюється тільки певним потоком), реалізація таких структур даних вирішує проблему. Знову ж таки, на жаль, таке вирішення можна застосовувати далеко не у всіх випадках (наприклад, у ситуації з банківськими рахунками різні потоки, зрештою, повинні якимось чином модифікувати загальний для всіх рахунок).

У всіх інших випадках потрібно забезпечувати захист змін від впливу інших потоків. Це і є основним завданням синхронізації. Перейдемо до аналізу різних підходів до його розв'язання.

5.2.2. Критичні секції та блокування

Поняття критичної секції

Розглянемо використання найпростішої ідеї для вирішення проблеми змагань. Неважко помітити, як джерелом нашої помилки є те, що зовні найпростіша операція покладення грошей на рахунок насправді розпадається на кілька операцій, при цьому завжди залишається шанс втручання між ними якогось іншого потоку. У цьому випадку кажуть, що вихідна операція не є атомарною.

Звідси випливає, що розв'язанням проблеми змагання є перетворення фрагмента коду, який спричиняє проблему, в атомарну операцію, тобто в таку, котра гарантовано виконуватиметься цілковито без втручання інших потоків. Такий фрагмент коду називають *критичною секцією* (critical section):

```
// початок критичної секції
total_amount = total_amount + new_amount;
// кінець критичної секції
```

Тепер, коли два потоки візьмуться виконувати код критичної секції одночасно, той з них, що почав першим, виконає весь її код цілком до того, як другий почне своє виконання (другий потік чекатиме, поки перший не закінчить виконання коду критичної секції). У результаті підсумку гарантовано матимемо в нашій програмі послідовність подій за варіантом 2, і змагання не відбудеться ніколи.

Розглянемо властивості, які повинна мати критична секція.

- Взаємного виключення (mutual exclusion): у конкретний момент часу код критичної секції може виконувати тільки один потік.
- Прогресу: якщо кілька потоків одночасно запросили вхід у критичну секцію, один із них повинен обов'язково у неї ввійти (вони не можуть всі заблокувати один одного).
- Обмеженості очікування: процес, що намагається ввійти у критичну секцію, рано чи пізно обов'язково в неї ввійде.

Залишається відповісти на далеко не просте запитання: «Як нам змусити систему сприймати кілька операцій як одну атомарну операцію?»

Найпростішим розв'язанням такої задачі було б заборонити переривання на час виконання коду критичної секції. Такий підхід, хоча й розв'язує задачу в принципі, на практиці не може бути застосовуваний, оскільки внаслідок зациклення або аварії програми у критичній секції вся система може залишитися із заблокованими перериваннями, а отже, у непрацездатному стані.

Блокування

Раціональнішим розв'язанням є використання *блокувань* (locks). Блокування — це механізм, який не дозволяє більш як одному потокові виконувати код критичної секції. Використання блокування зводиться до двох дій: запровадження (заблокування, функція `acquire_lock()`) і зняття блокування (розблокування, функція `release_lock()`). У разі заблокування перевіряють, чи не було воно вже зроблене іншим потоком, і якщо це так, цей потік переходить у стан очікування, інакше він запроваджує блокування і входить у критичну секцію. Після виходу із критичної секції потік знімає блокування.

```
acquire_lock(lock);
// критична секція
release_lock(lock);
```

Так реалізовують властивість взаємного виключення, звідси походить інша назва для блокування — *м'ютекс* (mutex, скорочення від mutual exclusion). А втім, частіше ця назва позначає конкретний механізм ОС, що реалізує блокування, у такому смислі вона буде використана й у цій книзі.

Проблеми із реалізацією блокувань

Розглянемо наївну реалізацію критичної секції із використанням *змінних блокування*. З кожною критичною секцією пов'язують цілочислову змінну, якій присвоюють одиницю під час заблокування і нуль — після розблокування. Ось який приблизно має вигляд код такої реалізації:

```
int lock = 0;
void acquire_lock(int lock) {
    // якщо блокування немає (lock == 0), запровадити його (задати lock=1)
    // і вийти – ми ввійшли у критичну секцію
    // інакше чекати, поки блокування не знімуть
    while (lock != 0);  // (1)
    lock = 1;           // (2)
}
void release_lock(int lock) {
    // зняти блокування
    lock = 0;
}
```

Головна проблема цієї реалізації полягає в тому, що перевірка значення змінної блокування (1) та її зміна (2) не є частинами однієї атомарної операції. Коли інший потік перевірить значення змінної між цими двома операціями і виявить, що вона дорівнює 0, два потоки можуть опинитися у критичній секції одночасно.

Апаратна підтримка блокувань

Є низка алгоритмів, які дають можливість коректно реалізувати блокування, спираючись на атомарність звичайних операцій записування в пам'ять і читання з пам'яті. До них належать алгоритми Деккера і Петерсона та алгоритм булочної (bakery algorithm). Із цими алгоритмами можна ознайомитися в літературі [41], [44], тут же обмежимося тими алгоритмами, які спираються на наявну апаратну підтримку. Розглянемо, як виглядає така апаратна підтримка і що можна реалізувати на її основі.

Для організації блокування в архітектурі IA-32 може бути використана спеціальна інструкція процесора, яку називають *«перевірити і заблокувати»* (Test & Set Lock, TSL). Параметром цієї інструкції є деяка адреса в пам'яті (наприклад, яка визначає місцезнаходження змінної блокування). Розглянемо, що відбувається під час виконання цієї інструкції (для простоти вважатимемо, що в пам'яті, з якою працює ця інструкція, може зберігатися тільки 0 або 1, при цьому 1 означає «блокування є», 0 — «блокування відсутнє»).

1. Зчитуються дані, які перебувають у цей момент у пам'яті (поточний стан змінної блокування).
2. У пам'ять записують значення 1 (запроваджують блокування, якщо цього не було зроблено раніше, інакше значення змінної блокування не міняють).
3. Зчитані на кроці 1 дані повертає операція. Якщо блокування до початку виконання цієї інструкції не було запроваджене, операція поверне 0, у противному разі — 1.

Усі три кроки інструкції TSL виконуються як одна атомарна операція, це має гарантувати апаратне забезпечення.

Програмно-апаратна реалізація блокувань

Заблокування і розблокування із використанням інструкції TSL (тут буде використано псевдокод, у якому TSL(lock) відповідатиме виконанню операції TSL над ділянкою пам'яті, що відповідає змінній lock) можна реалізувати так:

```
void acquire_lock(int lock) {
    // якщо блокування немає (lock == 0), запровадити його (задати lock=1)
    // і вийти — ми ввійшли у критичну секцію
    // інакше чекати, поки блокування не знімуть, і не міняти lock
    while (TSL(lock) != 0);
}
void release_lock(int lock) {
    // зняти блокування
    lock = 0;
}
```

Головним недоліком описаної технології є те, що потік повинен постійно перевіряти в циклі, чи не зняли блокування. Таку ситуацію називають *активним очікуванням* (busy waiting), а таке блокування — *спін-блокуванням* (spinlock). Використання циклу під час активного очікування завантажує процесор і допустиме тільки упродовж короткого часу (це справедливо для однопроцесорних систем, у багатопроцесорних використання спін-блокувань може бути виправдане тому, що під час виконання циклу активного очікування одним процесором інші можуть продовжувати свою роботу).

Альтернативою активному очікуванню є операція *призупинення потоку* (thread blocking, sleep). При цьому потік за умови блокування негайно або через якийсь час переходить зі стану виконання у стан очікування, передаючи процесор іншому потокові. Для того щоб реалізувати критичну секцію за допомогою цієї операції, потрібно доповнити її операцією виведення потоку зі стану очікування (*поновленням потоку*, wakeup). Робота із критичною секцією в цьому разі буде мати такий вигляд:

```
int lockguard = 0;   // блокування для критичних секцій усередині
                     // acquire_lock() і release_lock()
void acquire_lock(int lock) {
    // перевірити, чи безпечно виконувати цю дію
    while (TSL(lockguard) != 0);
    // якщо блокування запроваджене – призупинити потік
    if (lock == 1) sleep();
    // інакше заблокувати й увійти у критичну секцію
    else lock = 1;
    // дає можливість іншим потокам виконувати цю дію
    // цей код повинен виконуватися завжди у разі завершення операції
    lockguard = 0;
}
void release_lock(int lock) {
    // перевірити на безпеку виконання цієї дії
    while (TSL(lockguard) != 0);
    // розбудити один із призупинених потоків, якщо вони є
    if (waiting_threads()) wakeup(some_thread);
    // якщо призупинені потоки відсутні – зняти блокування
    else lock = 0;
    // дає можливість іншим потокам виконувати цю дію
    // цей код повинен виконуватися завжди у разі завершення операції
    lockguard = 0;
}
```

Виділимо деякі особливості цього алгоритму.

- Інструкцію TSL використовують тільки для забезпечення атомарності виконання операцій acquire_lock і release_lock (для перевірки безпеки цих операцій використовують змінну lockguard). У результаті потік перебуватиме у стані активного очікування не довше, ніж потрібно іншим потокам для виконання цих операцій.
- За наявності блокування потік буде призупинено. На практиці зазвичай із блокуванням lock пов'язують чергу очікування, в яку й буде поміщено цей потік.
- У разі спроби розблокування, якщо відповідна черга очікування непорожня, виконання одного із призупинених потоків буде продовжене і блокування не зняте. Реальне розблокування відбудеться тільки тоді, коли черга очікування порожня.
- Псевдокод неповністю відображає ту ситуацію, що змінній lockguard після виходу з acquire_lock і release_lock повинне бути присвоєне значення 0 завжди (навіть під час перемикання контексту). У противному разі інші потоки не змогли б виконувати ці операції.

Призупинення й поновлення потоку спричиняють перемикання контексту, що є досить тривалою операцією. На практиці ці примітиви часто поєднують зі спін-блокуванням, для чого приблизно оцінюють час, необхідний для перемикання контексту (ціну призупинення), і перед тим, як призупинити потік, виконують спін-блокування впродовж цього часу.

У результаті в найкращому разі вхід у критичну секцію займе набагато менше часу, ніж було б витрачено на перемикання контексту, у гіршому – буде затрачено часу всього удвічі більше, ніж під час негайного призупинення потоку.

Механізми синхронізації ОС

Описані алгоритми реалізації блокувань дають нам можливість краще зрозуміти їхню природу. З іншого боку, на практиці доцільніше використовувати готові вирішення, і було б дуже зручно, якби їх надавала сама операційна система. Такі вирішення працюють ефективніше і надійніше, можуть користуватися перевагами роботи в режимі ядра, краще документовані, поряд із конструкціями низького рівня (наприклад, блокуваннями) можуть реалізовувати синхронізацію на більш високому рівні (надаючи складніші об'єкти).

Сучасні ОС надають широкий набір готових механізмів синхронізації, які будуть розглянуті в наступному розділі.

5.3. Базові механізми синхронізації потоків

Механізмами синхронізації є засоби операційної системи, які допомагають розв'язувати основне завдання синхронізації – забезпечувати координацію потоків, котрі працюють зі спільно використовуваними даними. Якщо такі засоби – це мінімальні блоки для побудови багатопотокових програм, їх називають *синхронізаційними примітивами*.

Синхронізаційні механізми поділяють на такі основні категорії:

- універсальні, низького рівня, які можна використовувати різними способами (*семафори*);
- прості, низького рівня, кожен з яких пристосований до розв'язання тільки однієї задачі (*м'ютекси* та *умовні змінні*);
- універсальні високого рівня, виражені через прості; до цієї групи належить концепція *монітора*, яка може бути виражена через м'ютекси та умовні змінні;
- високого рівня, пристосовані до розв'язання конкретної синхронізаційної задачі (блокування читання-записування і бар'єри).

Розглянемо різні механізми і дамо оцінку перевагам, які має кожен з них, а також можливі їхні недоліки.

Для подальшої ілюстрації матеріалу цього розділу візьмемо класичний приклад, який демонструє необхідність синхронізації – задачу виробників-споживачів (producer-consumer problem) або задачу обмеженого буфера (bounded buffer).

Формулювання задачі просте. Припустимо, що в нас є потоки-виробники і потоки-споживачі. Виробник створює об'єкти, споживач їх отримує. Завданням є так синхронізувати їхню роботу, щоб споживач не міг намагатися отримувати ще не

створені об'єкти, а виробник не міг зробити більше об'єктів, ніж зможе отримати споживач.

Для синхронізації потоків помістимо між виробником і споживачем буфер фіксованої довжини *n*. Виробник може поміщати об'єкти у буфер, споживач – забирати їх звідти. Якщо споживач забирає об'єкт, його вилучають із буфера. Необхідно забезпечити кілька вимог.

1. Коли виробник або споживач працює із буфером, решта потоків повинна чекати, поки він завершить свою роботу.
2. Коли виробник намагається помістити об'єкт у буфер, а буфер повний (у ньому *n* об'єктів), він має дочекатися, поки в ньому з'явиться місце.
3. Коли споживач намагається забрати об'єкт із буфера, а буфер порожній, він має дочекатися, поки в ньому з'явиться об'єкт.

5.3.1. Семафори

Концепцію семафорів запропонував у 1965 році Е. Дейкстра – відомий голландський фахівець у галузі комп'ютерних наук. Семафори є найстарішими синхронізаційними примітивами з числа тих, які застосовуються на практиці.

Семафор – це спільно використовуваний невід'ємний цілочисловий лічильник, для якого задано початкове значення і визначено такі атомарні операції.

- *Зменшення семафора* (down): якщо значення семафора більше від нуля, його зменшують на одиницю, якщо ж значення дорівнює нулю, цей потік переходить у стан очікування доти, поки воно не стане більше від нуля (кажуть, що потік «очікує на семафорі» або «заблокований на семафорі»). Цю операцію називають також *очікуванням* – wait. Ось її псевдокод:

```
void down (semaphore_t sem) {
    if (sem > 0) sem--;
    else sleep();
}
```

- *Збільшення семафора* (up): значення семафора збільшують на одиницю; коли при цьому є потоки, які очікують на семафорі, один із них виходить із очікування і виконує свою операцію down. Якщо на семафорі очікують кілька потоків, то внаслідок виконання операції up його значення залишається нульовим, але один із потоків продовжує виконання (у більшості реалізацій вибір цього потоку буде випадковим). Цю операцію також називають *сигналізацією* – post. Ось її псевдокод:

```
void up (semaphore_t sem) {
    sem++;
    if (waiting_threads()) wakeup (some_thread);
}
```

Фактично значення семафора визначає кількість потоків, що може пройти через цей семафор без блокування. Коли для семафора задане нульове початкове значення, то він блокуватиме всі потоки доти, поки якийсь потік його не «відкриє», виконавши операцію up. Операції up і down можуть бути виконані будь-якими потоками, що мають доступ до семафора.

Дейкстра використовував для операції down позначення P (від голландського probieren – перевіряти), а для операції up – позначення V (від голландського verhoren – збільшувати). Ці позначення часто використовують у літературі.

Особливості використання семафорів

Семафори можна використовувати для розв'язання двох різних задач синхронізації.

1. За їхньою допомогою можна організовувати взаємне виключення (захищати код критичних секцій від виконання більш як одним потоком). Найзручніше для цього використати *двійковий семафор*, який може приймати два значення: 0 і 1. Ось приклад реалізації критичної секції з використанням двійкового семафора:

```
semaphore_t sem = 1;  // на початку семафор відкритий
down (sem);
// критична секція
up (sem);
```

2. За їхньою допомогою можна організовувати очікування виконання деякої умови. Припустимо, що треба організувати очікування одним потоком завершення виконання іншого (аналог операції приєднання). У цьому випадку можна *використати семафор із початковим значенням 0 (закритий). Потік, який чекатиме*, повинен виконати для цього семафора операцію down, а щоб просигналізувати про завершення потоку, у його функції завершення потрібно для того самого семафора виконати up. Ось псевдокод (this_thread означає поточний потік):

```
void thread_init() {
    this_thread.sem = 0;  // на початку виконання потоку семафор закритий
}
void thread_exit() {
    up(this_thread.sem);  // розбудити потік, що очікує, якщо такий є
}
void thread_join(thread_t thread) {
    down(thread.sem);     // очікування завершення потоку thread
}
```

Реалізація задачі виробників-споживачів за допомогою семафорів

Спочатку назвемо синхронізаційні дії, потрібні для розв'язання цієї задачі.

1. Помістити операції безпосередньої зміни буфера (для виробника – поміщення об'єкта у буфер, для споживача – вилучення об'єкта із буфера) у критичні секції.
2. Організувати очікування відповідно до вимоги 2 (очікування виробника у разі повного буфера). При цьому споживач повинен повідомляти виробникам, які очікують, про те, що він забрав об'єкт із буфера (тобто буфер став неповним, якщо був повним).
3. Організувати очікування відповідно до вимоги 3 (очікування споживача у разі порожнього буфера). При цьому виробник повинен повідомляти споживачів, які очікують, про те, що він додав новий об'єкт у буфер (тобто буфер став непорожнім, якщо був порожнім).

Тепер розглянемо синхронізаційні примітиви, які нам потрібні. Кожна синхронізаційна операція потребує окремого семафора.

1. Для організації критичної секції використаємо двійковий семафор, як це робили раніше. Назвемо його lock. Він використовуватиметься як виробником, так і споживачем, захищаючи доступ до буфера від інших потоків (знову таки, як виробників, так і споживачів).
2. Для організації очікування виробника у разі повного буфера нам знадобиться семафор, поточне значення якого дорівнює кількості вільних місць у буфері. Назвемо його empty_items. Виробник перед спробою додати новий об'єкт у буфер зменшує цей семафор, переходячи в стан очікування, якщо той дорівнював 0. Споживач після того, як забере об'єкт із буфера, збільшить семафор, повідомивши таким способом про це виробників, що очікують (і розбудивши одного з них).
3. Для організації очікування споживача у разі порожнього буфера знадобиться семафор, поточне значення якого дорівнює кількості зайнятих місць у буфері. Назвемо його full_items. Споживач перед спробою забрати об'єкт із буфера зменшує цей семафор, переходячи у стан очікування, якщо той дорівнював 0. Виробник після того, як додасть об'єкт у буфер, збільшить семафор, повідомивши таким способом про це споживачів, що очікують (і розбудивши одного з них).

Ось псевдокод розв'язання цієї задачі:

```
semaphore_t lock = 1;        // для критичної секції
semaphore_t empty_items = n; // 0 – буфер повний, від початку він порожній
semaphore_t full_items = 0;  // якщо 0 – буфер порожній

// виробник
void producer(){
    item_t item = produce();  // створити об'єкт
    down(empty_items); // чи є місце для об'єкта?
    down(lock);        // вхід у критичну секцію
    append_to_buffer(item);      // додати об'єкт item у буфер
    up(lock);          // вихід із критичної секції
    up(full_items);    // повідомити споживачів, що є новий об'єкт
}

// споживач
void consumer() {
    item_t item;
    down(full_items);  // чи не порожній буфер?
    down(lock);        // вхід у критичну секцію
    item = receive_from_buffer();  // забрати об'єкт item з буфера
    up(lock);          // вихід із критичної секції
    up(empty_items);   // повідомити виробників, що є місце
    consume(item);     // спожити об'єкт
}
```

Проблеми використання семафорів

Як бачимо, розв'язання задачі виробників-споживачів ілюструє обидва способи використання семафорів: для взаємного виключення і для очікування. Насправді

така універсальність має свій зворотний бік: код із використанням семафорів виходить досить складним.

- Далеко не завжди просто відстежити, де і як може змінитися значення семафора (одні зміни можуть відбуватися в одних фрагментах коду, інші — у зовсім інших і т. ін.).
- Оскільки всі синхронізаційні дії реалізовані семафорами, програму потрібно добре документувати, щоб було ясно, який семафор за що відповідає.

Ось приклад того, як легко зробити помилку в такому коді. Припустимо, що ми поміняли місцями `down(empty_items)` і `down(lock)` в `producer()`, тому перевірка умови з можливим очікуванням буде зроблена *всередині* критичної секції. У цьому разі ми можемо отримати таку послідовність дій, якщо буфер повний.

1. Виробник успішно входить у критичну секцію всередині `producer()`, закриваючи семафор `lock`.
2. Виробник перевіряє, чи не заповнений буфер даних, і блокується на семафорі `empty_items`. Семафор `lock` залишається закритим.
3. Споживач намагається ввійти у критичну секцію всередині `consumer()` і блокується на семафорі `lock`.

У цьому стані обидва потоки й залишаються — це *взаємне блокування* (deadlock), коли кожен із потоків очікує на блокуванні, запровадженим іншим (докладніше познайомимося з такими проблемами у розділі 7).

Для розв'язання деяких із цих проблем було запропоновано розділити дві функції семафорів, надавши кожній із них окремий синхронізаційний примітив. Для організації взаємного виключення використовують м'ютекси, очікування виконання умови — умовні змінні. У наступних розділах ми познайомимося з цими примітивами.

5.3.2. М'ютекси

Поняття м'ютекса багато в чому збігається з поняттям блокування, визначеним у розділі 5.2. *М'ютексом* називають синхронізаційний примітив, що не допускає виконання деякого фрагмента коду більш як одним потоком. Фактично м'ютекс є реалізацією блокування на рівні ОС.

М'ютекс, як і випливає з його назви, реалізує взаємне виключення. Його основне завдання — блокувати всі потоки, які намагаються отримати доступ до коду, коли цей код уже виконує деякий потік.

М'ютекс може перебувати у двох станах: вільному і зайнятому. Початковим станом є «вільний». Над м'ютексом можливі дві атомарні операції.

- *Зайняти м'ютекс* (`mutex_lock`): якщо м'ютекс був вільний, він стає зайнятим, і потік продовжує своє виконання (входячи у критичну секцію); якщо м'ютекс був зайнятий, потік переходить у стан очікування (кажуть, що потік «очікує на м'ютексі», або «заблокований на м'ютексі»), виконання продовжує інший потік. Потік, який зайняв м'ютекс, називають *власником м'ютекса* (mutex owner):

```
mutex_lock (mutex_t mutex) {
    if (mutex.state == free) {
        mutex.state = locked;
```

```
        mutex.owner = this_thread;
    }
    else sleep();
}
```

- *Звільнити м'ютекс* (mutex_unlock): м'ютекс стає вільним; якщо на ньому очікують кілька потоків, з них вибирають один, він починає виконуватися, займає м'ютекс і входить у критичну секцію. У більшості реалізацій вибір потоку буде випадковим. Звільнити м'ютекс може тільки його власник. Ось псевдокод цієї операції:

```
mutex_unlock (mutex_t mutex) {
    if (mutex.owner != this_thread) return error;
    mutex.state = free;
    if (waiting_threads()) wakeup (some_thread);
}
```

Деякі реалізації надають ще третю операцію: *спробувати зайняти м'ютекс* (mutex_trylock): якщо м'ютекс вільний, діяти аналогічно до mutex_lock, якщо зайнятий — негайно повернути помилку і продовжити виконання.

Ось найпростіша реалізація критичної секції за допомогою м'ютекса.

```
mutex_t mutex;
mutex_lock(mutex);
// критична секція
mutex_unlock(mutex);
```

Основною відмінністю м'ютексів від двійкових семафорів (семафори цього виду ми використовували для блокування), є те, що звільнити м'ютекс може тільки його власник, тоді як змінити значення семафора може будь-який потік, котрий має до нього доступ. Ця відмінність досить суттєва і робить реалізацію взаємних виключень за допомогою м'ютексів простішою (з коду завжди ясно, який потік може змінити стан м'ютекса).

Правила спрощеного паралелізму

Правила спрощеного паралелізму (easy concurrency rules) [78] призначені для спрощення програмування на базі м'ютексів. Вони ґрунтуються на тому очевидному факті, що м'ютекс захищає не код критичної секції, а спільно використовувані дані всередині цієї секції.

- Кожна змінна, яку спільно використовує більш як один потік, має бути захищена окремим м'ютексом (скільки змінних, стільки м'ютексів):

```
volatile int i, data[100];
mutex_t i_mutex, data_mutex;
```

- Перед кожною операцією зміни такої змінної відповідний м'ютекс має бути зайнятий, а після зміни звільнений:

```
mutex_lock(i_mutex); i++; mutex_unlock(i_mutex);
```

- Якщо треба працювати одночасно із кількома спільно використовуваними змінними, необхідно зайняти всі їхні м'ютекси до початку роботи і звільнити їх тільки після повного закінчення роботи. Цим роботу розділяють на три етапи:

```
// зайняття м'ютексів
mutex_lock(i_mutex); mutex_lock(data_mutex);
```

```
// робота зі змінними
data[i++] = 100;
// звільнення м'ютексів
mutex_unlock(i_mutex); mutex_unlock(data_mutex);
```

Зазначимо, що для спільно використовуваних даних задано клас пам'яті volatile. Використання такого модифікатора сповіщає компілятор мови С, про можливе асинхронне змінення змінної поза даною програмою. Це завжди потрібно робити для спільно використовуваних даних.

Реалізація задачі виробників-споживачів за допомогою м'ютексів

З'ясуємо можливість розв'язання задачі виробників-споживачів за допомогою самих лише м'ютексів та розглянемо проблеми, які з цим пов'язані. Тут не наводитимемо повного коду розв'язання завдання, зупинимося тільки на ключових моментах.

Проблем із реалізацією критичних секцій не виникне, оскільки в цьому разі м'ютекси використовуватимуться за прямим призначенням.

Розглянемо організацію очікування виконання умови. Ця задача досить складна, оскільки м'ютекс, з одного боку, має тільки два стани, а з іншого — не може бути зайнятий в одному потоці, а звільнений в іншому. Це змушує вдатися до активного очікування, перевіряючи в циклі, чи не стала умова істинною (при цьому займаючи м'ютекс на час перевірки і звільняючи після неї). Ось як можна реалізувати функцію consumer():

```
mutex_t lock;
void consumer() {
    item_t item;
    for (;;) {  // нескінченний цикл активного очікування
                // вихід із нього, якщо буфер не порожній
        mutex_lock(lock);           // вхід у критичну секцію
        if (count_items_in_buffer() != 0) {   // якщо буфер не порожній
            item = receive_from_buffer();  // забрати об'єкт із буфера
            mutex_unlock(lock);     // вийти із критичної секції
            break;                  // вийти з циклу
        }
        mutex_unlock(lock);         // вихід із критичної секції
    }
    consume(item);      // спожити об'єкт
}
```

Реалізація функції producer() загалом аналогічна. Очевидно, що м'ютекси не підходять для організації умовного очікування. Потрібен примітив, який дасть можливість потоку переходити у стан очікування до настання деякої події. Таким примітивом є умовна змінна, яку розглянемо в наступному розділі.

Рекурсивні м'ютекси

Розглянемо випадок, коли потік, який займає м'ютекс, спробує зайняти той самий м'ютекс іще раз. Для звичайних м'ютексів отримаємо взаємне блокування, оскільки потік очікуватиме на м'ютексі, що його звільнити може тільки він сам.

Рекурсивний м'ютекс — особливий вид м'ютекса. Він дозволяє повторне зайняття тим самим потоком, а також відстежує, який потік намагається його зайняти.

Коли це не той потік, що його вже займає, м'ютекс поводиться як звичайний, і потік переходить у стан очікування. Коли ж це той самий потік, внутрішній лічильник блокувань цього м'ютекса збільшують на одиницю, і потік продовжує своє виконання. У разі звільнення м'ютекса внутрішній лічильник зменшують на одиницю, для інших потоків рекурсивний м'ютекс буде розблоковано тільки тоді, коли лічильник дійде до нуля (тобто коли всі його блокування одним потоком будуть зняті).

Рекурсивні м'ютекси менш ефективні в реалізації, не можуть бути використані разом з умовними змінними (це питання розглянемо в наступному розділі), тому звертатися до них потрібно тільки тоді, коли без цього не можна обійтися. Наприклад, бібліотека функцій може використовувати такі м'ютекси для того, щоб уникнути взаємних блокувань під час повторного виклику таких функцій одним потоком.

Реалізація м'ютексів у POSIX

М'ютекси у POSIX зображаються структурами даних, що мають тип pthread_mutex_t. Для ініціалізації м'ютекса найпростіше використати статичний ініціалізатор:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Для того щоб зайняти м'ютекс, передбачено функцію pthread_mutex_lock(), а для звільнення — pthread_mutex_unlock(). Ось приклад роботи із м'ютексом для потоків POSIX:

```
#include<pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex);  // якщо тут м'ютекс зайнятий — очікування
// тут цей потік зайняв (заблокував) м'ютекс
pthread_mutex_unlock(&mutex);
```

Неблокувальним аналогом функції pthread_mutex_lock() є pthread_mutex_trylock(). У разі неможливості зайняти м'ютекс вона негайно повертає помилку з кодом EBUSY.

5.3.3. Умовні змінні та концепція монітора

Поняття умовної змінної

Умовною змінною називають синхронізаційний примітив, який дає змогу організувати очікування виконання умови всередині критичної секції, заданої м'ютексом. Умовна змінна завжди пов'язана із конкретним м'ютексом і даними, захищеними цим м'ютексом. Для умовної змінної визначено такі операції.

- *Очікування* (wait). Додатковим вхідним параметром ця операція приймає м'ютекс, який повинен перебувати в закритому стані. Виклик wait відбувається в ситуації, коли не виконується деяка умова, потрібна потоку для продовження роботи. Внаслідок виконання wait потік (позначатимемо його T_W) припиняється (кажуть, що він «очікує на умовній змінній»), а м'ютекс відкривається (ці дві дії відбуваються атомарно). Так інші потоки отримують можливість увійти в критичну секцію і змінити там дані, які вона захищає, можливо, виконавши умову, потрібну потоку T_W. На цьому операція wait не завершується —

її завершить інший потік, викликавши операцію signal після того, як умову буде виконано.

- *Сигналізація* (signal). Цю операцію потік (назвемо його T_S) має виконати після того, як увійде у критичну секцію і завершить роботу з даними (виконавши умову, яку очікував потік, що викликав операцію wait). Ця операція перевіряє, чи немає потоків, які очікують на умовній змінній, і якщо такі потоки є, переводить один із них (T_W) у стан готовності (цей потік буде поновлено, коли відповідний потік T_S вийде із критичної секції). Внаслідок поновлення потік T_W завершує виконання операції wait – блокує м'ютекс знову (поновлення і блокування теж відбуваються атомарно). Якщо немає жодного потоку, який очікує на умовній змінній, операція signal не робить нічого, і інформацію про її виконання в системі не зберігають.
- *Широкомовна сигналізація* (broadcast) відрізняється від звичайної тим, що переведення у стан готовності і, зрештою, поновлення виконують для всіх потоків, які очікують на цій умовній змінній, а не тільки для одного з них.

Отже, виконання операції wait складається з таких етапів: відкриття м'ютекса, очікування (поки інший потік не виконає операцію signal або broadcast), закриття м'ютекса.

По суті, це перша *неатомарна* операція, визначена для синхронізаційного примітива, але така відсутність атомарності цілком контрольована (завжди відомо, де потік T_W перейшов у стан очікування і що його з цього стану виведе).

Особливості виконання операцій над умовною змінною

Операцію wait викликають, коли деяка умова, необхідна потоку для продовження, не виконується. Умова повинна бути пов'язана зі спільно використовуваними даними. Перед викликом wait потрібно перевірити цю умову в циклі while

```
while (! condition_expr)        // вираз для умови
    wait(condition, mutex);
```

Розглянемо, чому в цій перевірці використовують цикл while, а не умовний оператор if. На перший погляд, використання while зайве – нам досить перевірити умову один раз, для чого можна скористатися й if.

Пояснимо, чому це не так. Після того, як потік T_S виконає операцію signal, потік T_W не запускається негайно, а переходить у стан готовності до виконання. Виконуватися він почне тоді, коли його вибере планувальник (звичайно це відбувається швидко). Проте за проміжок часу між викликом signal і початком виконання потоку T_W ще один потік (позначимо його T_X) може ввійти у критичну секцію (згадаймо, що м'ютекс у цей час поки що відкритий) і змінити в ній дані так, що умова знову перестане виконуватися. Тепер, якби виклик wait стояв за if, після виходу з wait потік T_W заблокував би м'ютекс і продовжив своє виконання, незважаючи на те, що умова все одно не виконується. Це помилка, яка практично налагодженню не підлягає (її називають *помилковим поновленням* – spurious wakeup). Коли ж використати while, то після виходу з wait умова знову буде перевірена, і якщо вона далі не виконується, потік T_W розблокує м'ютекс і знову перейде у стан очікування.

Використовуючи while у поєднанні з очікуванням на умовній змінній, ми гарантуємо, що потік продовжить свою роботу тільки під час виконання відповідної умови.

Наведемо приклад використання операції wait для організації очікування, поки звільниться місце у буфері (це фрагмент розв'язання задачі виробників-споживачів, яке розглянемо далі). Тут умовою є порожність буфера.

```
while (count_items_in_buffer() == n)   // поки буфер повний
    wait(not_full, mutex);             // not_full – умовна змінна
```

Операція signal повинна викликатися із критичної секції (тим паче, що параметром цієї операції теж є м'ютекс). Потік T_S має викликати signal, коли умова, якої очікував потік T_W, почне виконуватися.

```
// тут condition_expr == true
signal(condition, mutex);
```

Ось приклад використання операції signal для повідомлення про те, що буфер звільнився:

```
item = receive_from_buffer();   // звільнити буфер
signal(not_full, mutex);        // сигналізувати, що буфер непорожній
```

Розглядаючи цей фрагмент у поєднанні із попереднім, зауважимо: якщо якийсь потік вставить об'єкт у буфер між викликом signal і поновленням виконання потоку, це негайно зробить умову в циклі while знову істинною, внаслідок чого операція wait буде виконана знову.

Операція broadcast може виявитися корисною тоді, коли виконання умови має спричинити негайне розблокування всіх потоків, які очікують. Наприклад, якщо один потік записує у базу даних, а інші чекають, поки він закінчить це робити, щоб прочитати дані з неї (при цьому потоки-читачі один одному не заважають), є сенс після завершення записування виконати саме операцію broadcast, щоб усі читачі змогли негайно розпочати читання.

Відмінності умовних змінних від семафорів

Наведемо принципові відмінності умовних змінних від семафорів, які використовують для організації очікування за умовою.

- Умовні змінні можуть бути використані лише всередині критичних секцій, при цьому інші потоки можуть входити у критичну секцію під час очікування на умовній змінній. Семафори небезпечно використовувати всередині критичних секцій, оскільки це зазвичай призводить до взаємного блокування.
- Умовні змінні не зберігають стану, а семафори — зберігають. Ось що це означає.
  - Якщо потік T_S виконує signal для умовної змінної і потоки, які очікують на цій умовній змінній, відсутні, нічого не відбувається і жодної інформації в системі не зберігають. Коли потім потік T_W виконає операцію wait на цій умовній змінній, то він призупиниться.
  - Якщо потік T_S виконує up для семафора і потоки, які очікують на цьому семафорі, відсутні, значення семафора все одно збільшують і нове значення зберігають у системі. Коли потім потік T_W виконає операцію down для цього

семафора, він замість призупинення зменшить семафор і продовжить своє виконання (незважаючи на те, що потік T_S раніше виконав свою операцію намарно).

Рекурсивні м'ютекси й умовні змінні

Рекурсивні м'ютекси не можуть бути використані у поєднанні з умовними змінними, оскільки рекурсивний м'ютекс може не звільнитися разом із початком очікування всередині `wait`, якщо він перед цим був зайнятий кілька разів (а це гарантує взаємне блокування).

Очікування виконання кількох умов

Можна організовувати очікування виконання кількох умов, хоча це рекомендують робити тільки у разі необхідності, щоб не ускладнювати код. Для такого очікування потрібно використати одну умовну змінну й у циклі `while` перевіряти виконання кількох умов:

```
// повинно виконуватися condition_expr1 або condition_expr2
while (! condition_expr1 && ! condition_expr2)
      wait(condition, mutex);
// повинно виконуватися condition_expr1 і condition_expr2
while (! condition_expr1 || ! condition_expr2)
      wait(condition, mutex);
```

Кожну умову треба сигналізувати окремо:

```
// виконують condition_expr1
signal (condition, mutex);
// ...
// виконують condition_expr2
signal (condition, mutex);
```

Поняття монітора

Як ми бачимо, умовні змінні не використовують окремо від м'ютексів, причому є кілька правил взаємодії між цими примітивами. Ці правила є підґрунтям поняття *монітора* — синхронізаційної концепції вищого рівня.

Монітором називають набір функцій, які використовують один загальний м'ютекс і нуль або більше умовних змінних для керування паралельним доступом до спільно використовуваних даних відповідно до певних правил. Функції цього набору називають функціями монітора.

Ось правила, яких слід дотримуватися у разі реалізації монітора.

- Під час входу в кожну функцію монітора потрібно займати м'ютекс, під час виходу — звільняти. Отже, у кожний момент часу тільки один потік може перебувати всередині монітора (під яким розуміють сукупність усіх його функцій).
- Під час роботи з умовною змінною (і під час очікування, і під час сигналізації) необхідно завжди вказувати відповідний м'ютекс. Не можна працювати з умовними змінними, якщо м'ютекс незайнятий.
- Під час перевірки на виконання умови очікування потрібно використати цикл, а не умовний оператор.

Ідея монітора була вперше запропонована в 1974 році відомим ученим у галузі комп'ютерних наук Ч. А. Хоаром. Монітор часто розуміють як високорівневу конструкцію мови програмування (як приклад такої мови звичайно наводять Java), а саме як набір функцій або методів класу, всередині яких автоматично зберігається неявний загальний м'ютекс разом із операціями очікування і сигналізації. Насправді, як ми бачимо, концепція монітора може ґрунтуватися на базових примітивах — м'ютексах і умовних змінних — і не повинна бути обмежена якоюсь однією мовою.

Монітори Хоара відрізняються від тих, що були розглянуті тут (ці монітори ще називають *MESA-моніторами* за назвою мови, у якій вони вперше з'явилися). Головна відмінність полягає у реалізації сигналізації.

- У моніторах Хоара після сигналізації потік T_S негайно припиняють, і керування переходить до потоку T_W, який при цьому захоплює блокування. Коли потік T_W вийде із критичної секції або знову виконає операцію очікування, потік T_S буде поновлено.
- У MESA-моніторах, як було видно, після сигналізації потік T_S продовжує своє виконання, а потік T_W просто переходить у стан готовності до виконання. Він зможе продовжити своє виконання, коли потік T_S вийде з монітора (чекати цього доведеться недовго, тому що звичайно сигналізація відбувається наприкінці функції монітора).

Результатом є те, що для моніторів Хоара не обов'язково перевіряти умову очікування в циклі, досить умовного оператора (потік негайно отримує керування після виходу з очікування і не може статися так, що за цей час інший потік увійде в монітор і змінить умову). З іншого боку, ці монітори менш ефективні (потрібно витрачати час на те, щоб припиняти і поновлювати потік T_S); потрібно мати повну гарантію того, що між виконанням сигналізації та переданням керування потоку T_W планувальник не передасть керування іншому потокові T_X, який увійде у функцію монітора. Забезпечення такої гарантії потребує втручання в алгоритм роботи планувальника ОС.

Ці недоліки призводять до того, що на практиці використовують переважно MESA-монітори.

Ось приклад розв'язання задачі очікування завершення потоку із використанням монітора.

```
// всі функції перебувають у моніторі
// з кожним потоком пов'язана умовна змінна finished_cond
// і прапорець finished — спільно використовувані дані
void thread_init() {
    mutex_lock(mutex);
    // на початку виконання потоку умова не виконується
    this_thread.finished = 0;
    mutex_unlock(mutex);
}
void thread_exit() {
    mutex_lock(mutex);
    this_thread.finished = 1;
    // розбудити потік, який очікує, якщо він є
    signal(this_thread.finished_cond, mutex);
```

```
    mutex_unlock(mutex);
}
void thread_join(thread_t thread) {
    mutex_lock(mutex);
    // очікування закінчення потоку thread
    while (! thread.finished)
        wait(thread.finished_cond, mutex);
    mutex_unlock(mutex);
}
```

Реалізація задачі виробників-споживачів за допомогою монітора

Розглянемо синхронізаційні примітиви, які можуть знадобитися під час реалізації цієї задачі.

- Для того щоб забезпечити перебування в моніторі тільки одного потоку в конкретний момент часу, використовуватимемо м'ютекс lock. Він буде спільним для всіх функцій семафора, з ним працюватимуть як виробник, так і споживач.
- Для організації очікування виробника у разі повного буфера потрібна умовна змінна, сигналізація якої означатиме, що місце у буфері звільнилося. Назвемо цю змінну not_full. Перед спробою додати новий об'єкт у буфер виробник перевіряє, чи буфер повний і, якщо це так, виконує очікування на цій змінній. Споживач, забравши об'єкт із буфера, сигналізує not_full, повідомляючи цим про наявність вільного місця виробникам, які очікують (і перевівши у стан готовності одного з них).
- Для організації очікування споживача під час порожнього буфера потрібна умовна змінна, сигналізація якої означатиме, що у буфері з'явився об'єкт. Назвемо цю змінну not_empty. Перед спробою забрати об'єкт із буфера споживач перевіряє, чи буфер порожній і, якщо це так, виконує очікування на цій змінній. Виробник, додавши об'єкт у буфер, сигналізує not_empty, повідомляючи цим про наявність об'єктів споживачам, які очікують (і перевівши у стан готовності одного з них).

Завданнями функцій монітора буде забезпечення роботи із буфером. Для кращої організації коду запишемо ці функції окремо від коду виробника і споживача.

Ось псевдокод розв'язання цієї задачі:

```
mutex_t lock;          // для критичної секції
condition_t not_empty; // сигналізує про те, що буфер непорожній
condition_t not_full;  // сигналізує про те, що буфер неповний
int n = 100;           // максимально можлива кількість елементів

// виробник
void producer(){
    item_t item = produce();    // створити об'єкт
    put_into_buffer(item);
}

// споживач
void consumer() {
    item_t item = obtain_from_buffer();
    consume(item);       // спожити об'єкт
}
```

```
// функції монітора
void put_into_buffer(item_t item) {
    mutex_lock(lock);              // вхід у критичну секцію
    while (count_items_in_buffer() == n)
        wait (not_full, lock);     // чекати, поки буфер повний
                                   // на цей час зняти блокування
    append_to_buffer(item);        // додати об'єкт item у буфер
    signal(not_empty, lock);       // повідомити, що є новий об'єкт
    mutex_unlock(lock);            // вихід із критичної секції
}

item_t obtain_from_buffer () {
    item_t item;
    mutex_lock(lock);              // вхід у критичну секцію
    while (count_items_in_buffer() == 0)
        wait (not_empty, lock);    // чекати, поки буфер порожній
                                   // на цей час зняти блокування
    item = receive_from_buffer(); // забрати об'єкт item із буфера
    signal(not_full, lock);        // повідомити, що є вільне місце
    mutex_unlock(lock);            // вихід із критичної секції
    return item;
}
```

Цей код зрозуміліший, ніж код із використанням семафорів, насамперед тому, що не потрібно здогадуватися, що означає збільшення або зменшення того чи іншого семафора — відразу видно, який примітив відповідає за взаємне виключення, а який — за організацію очікування.

Деякі джерела (наприклад, [65]) взагалі не рекомендують користуватися семафорами для синхронізації, обмежуючи себе тільки моніторами (м'ютексами й умовними змінними). Треба, однак, зазначити, що:

- у деяких системах (наприклад, у Linux до появи NPTL) семафори — це єдиний засіб синхронізації потоків різних процесів;
- в інших системах (наприклад, у Win32 API) майже не підтримуються умовні змінні (принаймні, реалізувати їх там складно);
- існує великий обсяг коду, написаного із використанням семафорів, який може виявитися необхідним для читання і підтримки.

Отже, вивчення семафорів є необхідним.

Загальна стратегія організації паралельного виконання

Для коректної організації виконання багатопотокових програм особливо важливі два із розглянутих раніше правил.

- М'ютекс захищає не код критичної секції, а спільно використовувані дані всередині цієї секції.
- Виклик `wait` для умовної змінної відбувається тоді, коли не виконується умова, пов'язана зі спільно використовуваними даними всередині критичної секції, виклик `signal` — коли умова, пов'язана з цими даними, починає виконуватися.

Як бачимо, м'ютексами і умовними змінними керують дані, що вони захищають. Так само вся концепція монітора побудована навколо спільно використовуваних даних. У разі розробки на C++ є сенс надати самим спільно використовуваним даним право відповідати за свою синхронізацію перетворенням їх в об'єкти класів та інкапсуляцією всіх синхронізаційних примітивів у методах цих класів. Рекомендують такий базовий підхід до розробки багатопотокових програм на C++ [65].

1. Виділити одиниці паралельного виконання. Зробити кожну з них потоком. Потоки можуть бути інкапсульовані у класи з методом go(), який виконує функцію потоку.
2. Виділити спільно використовувані структури даних. Зробити кожну з таких структур класом. Виділити методи класів — дії, які потоки виконуватимуть із цими структурами даних.
3. Записати основний цикл виконання кожного потоку.

На цих трьох етапах ми поки що не займаємося синхронізацією — усе відбувається на більш високому рівні. Тепер для кожного класу потрібно виконати такі дії.

1. Визначити всі синхронізаційні дії, котрі необхідно виконувати з об'єктами цього класу. Визначити тип кожної дії: взаємне блокування або очікування умови.
2. Створити м'ютекси або умовні змінні для кожної дії.
3. Розробити методи класів, використовуючи для синхронізації ці м'ютекси і умовні змінні (звичайно ці методи роблять функціями монітора).

Реалізація умовних змінних у POSIX

Умовні змінні POSIX цілком відповідають наведеному опису. Для роботи з ними використовують тип pthread_cond_t. Для ініціалізації умовної змінної найпростіше скористатися статичним ініціалізатором

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

Очікування умовної змінної реалізовано функцією pthread_cond_wait(). Ось приклад використання цієї функції у поєднанні з м'ютексом:

```
#include<pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
// ...
pthread_mutex_lock(&mutex);
while (!cond_expression) pthread_cond_wait(&cond, &mutex);
// ...
pthread_mutex_unlock(&mutex);
```

Для сигналізації умовної змінної використовують функцію pthread_cond_signal(), для широкомовної сигналізації — pthread_cond_broadcast(). Обидві ці функції приймають покажчик на pthread_cond_t і повертають int:

```
// cond_expression == true
pthread_cond_signal (&cond);
pthread_cond_broadcast (&cond);
```

5.3.4. Блокування читання-записування

М'ютекси є засобом, який захищає спільно використовувані дані від будь-якого одночасного доступу з боку кількох потоків — будь то читання чи зміна. Насправді нам не завжди потрібен такий однозначний захист, наприклад, для певного типу задач хотілося б розрізняти читання спільно використовуваних даних та їхню модифікацію (для того, щоб, скажімо, дозволяти читання кільком потокам одночасно, а модифікацію — тільки одному). Для розв'язання такої задачі використовують *блокування читання-записування* (read-write locks).

Блокування читання-записування — це синхронізаційний примітив, для якого визначені два режими використання: відкриття для читання і відкриття для записування. При цьому повинні виконуватися такі умови:

- будь-яка кількість потоків може відкривати таке блокування для читання, коли немає жодного потоку, що відкрив його для записування;
- блокування може відкриватися для записування тільки за відсутності потоку, що відкрив його для читання або для записування.

Простіше кажучи, читати дані може будь-яка кількість потоків одночасно за умови, що ніхто ці дані не змінює; змінювати дані можна тільки тоді, коли їх ніхто не читає і не змінює.

Такі блокування корисні для даних, які зчитуються частіше, ніж модифікуються (наприклад, більшість СУБД реалізує блокування такого роду для забезпечення доступу до бази даних).

Типи блокувань

Розрізняють два типи блокувань читання-записування: *з кращим читанням* і *з кращим записом*. Відмінність між ними виявляється тоді, коли потік намагається відкрити таке блокування для читання за умови, що він робить це не першим і що є призупинені потоки, які очікують можливості відкрити це блокування для записування.

- У разі кращого читання потік негайно відкриває блокування для читання і продовжує свою роботу незалежно від того, є потоки-записувачі, що очікують, чи ні. Потоки-записувачі продовжують своє очікування.
- У разі кращого записування за наявності потоків-записувачів, що очікують, потік-читач припиняється й не буде поновлений доти, поки всі записувачі не виконають свої дії і не закриють блокування.

Зазначимо, що для обох типів потік-записувач не може відкрити блокування, поки його тримає відкритим хоча б один читач, — перевага надається тільки новим потокам-читачам, які намагаються відкривати додаткові блокування.

Операції блокувань

Розглянемо операції, допустимі для блокувань читання-записування.

- *Відкриття для читання* (`rwlock_rdlock`). Якщо є потік, який відкрив блокування для записування, поточний потік припиняють. Якщо такий потік відсутній:
  - у разі кращого читання блокування відкривають для читання і потік продовжує своє виконання;

- у разі кращого записування перевіряють, чи немає призупинених потоків, які очікують відкриття цього блокування для записування; якщо вони є — потік припиняють, якщо немає — блокування відкривають для читання і потік продовжує своє виконання.

При цьому необхідно, щоб кілька потоків могли відкрити блокування для читання, тому в разі, коли блокування вже відкрите для читання, для його нового відкриття збільшують внутрішній лічильник потоків-читачів.

- *Відкриття для записування* (`rwlock_wrlock`). Якщо є потік, який відкрив блокування для читання або записування, поточний потік припиняють; коли жодного такого потоку немає, блокування відкривають для записування і потік продовжує своє виконання.
- *Закриття* (`rwlock_unlock`). У разі наявності кількох потоків, які відкрили блокування для читання, воно залишається відкритим для читання, і внутрішній лічильник потоків-читачів зменшують на одиницю. Якщо блокування відкрите для читання тільки одним потоком (лічильник дорівнює одиниці) його знімають, якщо є потоки-записувачі, які очікують на цьому блокуванні, один із них поновлюють. Коли блокування відкрите для записування, його знімають, при цьому за наявності потоків-читачів, що очікують, всі вони поновлюються, а з потоків-записувачів, що очікують, поновлюють тільки один. Якщо очікують і читачі й записувачі, результат залежить від типу блокування (у разі кращого читання або якщо жодного записувача немає, поновлюють усіх читачів, а якщо читачі відсутні у разі кращого записування — поновлюють одного з записувачів.

Блокування читання-записування, як і м'ютекси, мають власника, тому не можна закрити блокування в потоці, який його не відкривав.

Ось псевдокод прикладу використання таких блокувань для синхронізації доступу до банківського рахунку:

```
// блокування для захисту рахунку
rwlock_t account_lock;
// сума на рахунку – спільно використовувані дані
volatile float amount;

// додавання на рахунок
void set_amount (float new_amount) {
    rwlock_wrlock(account_lock);      // відкриття для записування
    amount = amount + new_amount;
    rwlock_unlock(account_lock);
}

// перегляд рахунку
float get_amount () {
    float cur_amount;
    rwlock_rdlock(account_lock);      // відкриття для читання
    cur_amount = amount;
    rwlock_unlock(account_lock);
    return cur_amount;
}
```

Голодування

Під час використання блокувань читання-записування можуть виникати проблеми, пов'язані з типом блокування. Якщо система працює за умов значного навантаження, для блокувань із кращим читанням можлива така послідовність дій потоків.

1. Потік T_{R1} відкриває блокування для читання L.
2. Потік T_{W1} намагається відкрити блокування L для записування і призупиняється.
3. Потік T_{R2} відкриває блокування L для читання.
4. Потік T_{R1} виконує операцію закриття блокування, але блокування L залишається зайнятим T_{R2}.
5. Потік T_{R3} відкриває блокування L для читання.
6. Потік T_{R2} виконує операцію закриття блокування, але блокування L залишається зайнятим T_{R3} (і т. д.).

У результаті додаткові потоки-читачі захоплюватимуть блокування L до того, як його звільнять усі попередні читачі. Це означає, що L ніколи повністю не звільниться, і потік-записувач T_{W1} ніколи не отримає керування. Таку ситуацію, як і під час планування, називають голодуванням.

Для блокувань із кращим записуванням голодування потоку-записувача бути не може. У нашому випадку потік T_{R2} на кроці 3 призупиниться і після закриття блокування потоком T_{R1} на кроці 4 потік T_{W1} дістане можливість виконуватися. З іншого боку, у цьому випадку за певних умов можна отримати голодування потоків-читачів.

Остаточного розв'язання проблеми голодування не запропоновано, рекомендують ретельно зважувати можливість використання різних типів блокувань і розраховувати можливість навантаження, яке може призвести до голодування.

Блокування читання-записування відповідно до POSIX

У літературі є багато прикладів реалізації блокувань читання-записування на базі інших примітивів, наприклад м'ютексів та умовних змінних. Останнім часом бібліотеки підтримки потоків (наприклад, NPTL у Linux) пропонують такі блокування як базові синхронізаційні примітиви; вони визначені стандартом POSIX (а нестандартні розширення дозволяють задавати і тип блокування).

Розглянемо реалізацію блокувань читання-записування відповідно до POSIX.

Для роботи із такими блокуваннями у стандарті POSIX використовують тип `pthread_rwlock_t`. Усі функції роботи із блокуваннями як параметр приймають покажчик на змінну такого типу. Найпростішу ініціалізацію блокування здійснюють за допомогою статичного ініціалізатора

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Відкриття такого блокування для записування відбувається за допомогою функції `pthread_rwlock_wrlock()`, відкриття для читання — за допомогою функції `pthread _rwlock_rdlock()`.

```
pthread_rwlock_wrlock(&rwlock);
pthread_rwlock_rdlock(&rwlock);
```

Для закриття блокування використовують функцію `pthread_rwlock_unlock()`:

```
pthread_rwlock_unlock(&rwlock);
```

5.3.5. Синхронізація за принципом бар'єра

Іноді буває зручно розбити задачу (наприклад, складний розрахунок) на кілька послідовних етапів, при цьому виконання наступного етапу не може розпочатися без *цілковитого* завершення попереднього. Якщо на кожному етапі роботу розподілити по окремих потоках, які паралельно виконуватимуть кожен свою частину, виникає запитання: яким чином дочекатися завершення всіх потоків попереднього етапу до початку наступного?

Звичайне використання `thread_join()` тут не підходить, тому що приєднати можна якийсь конкретний потік, а не цілий набір потоків. Для розв'язання цього завдання запропоновано концепцію спеціального синхронізаційного об'єкта — бар'єра [44].

Бар'єр — це блокування, яке зберігають доти, поки кількість потоків, що очікують, не досягне деякого наперед заданого числа, після чого всі ці потоки продовжують своє виконання.

Так, якщо наприкінці етапу задачі поставити бар'єр, він буде утримувати потоки від продовження, поки вони всі не завершать роботу цього етапу.

Операції бар'єра

Розглянемо дві операції бар'єра.

- *Ініціалізація бар'єра* (`barrier_init`). Параметром даної операції є n — кількість потоків, які ми синхронізуватимемо на цьому бар'єрі. Внутрішній лічильник призупинених потоків покладають рівним нулю. Значення n після виклику `barrier_init` міняти не можна.
- *Очікування на бар'єрі* (`barrier_wait`). Внаслідок даної операції лічильник призупинених потоків збільшується на одиницю. Якщо він не досягнув n, цей потік призупиняють (кажуть, що він «очікує на бар'єрі»). Якщо ж після збільшення лічильник виявляється рівним n (досягнуто максимальної кількості потоків для бар'єра), усі потоки, що очікували на бар'єрі, поновлюються (ситуація *переходу бар'єра*), при цьому одному з них повертають спеціальне значення (зазвичай –1), а іншим — нуль. Надалі потік, що отримав це спеціальне значення, може зробити деякі підсумкові дії після переходу бар'єра. При переході бар'єра його «скидають» (внутрішній лічильник призупинених потоків знову покладають рівним 0) і він починаючи із наступної операції `barrier_wait` блокуватиме потоки знову.

Ось псевдокод прикладу використання бар'єра:

```
barrier_t barrier_step;          // бар'єр етапу
void thread_fun() {              // функція потоку, розбита на етапи
    int res;
    step1();  // виконання дій етапу 1
    res = barrier_wait(barrier_step); // очікування завершення етапу 1
    if (res == -1) step1_done();
    step2();  // виконання дій етапу 2
    res = barrier_wait(barrier_step); // очікування завершення етапу 2
    if (res == -1) step2_done();
    // тощо
}
```

```
// n – кількість потоків для виконання поетапної задачі
void run () {
    thread_t threads[n];
    barrier_init(barrier_step, n);   // бар'єр для n потоків
    for (i=0; i<n; i++)
        threads[i] = thread_create(thread_fun);  // створюємо n потоків
    for (i=0; i<n; i++)
        thread_join(threads[i]);     // очікуємо завершення n потоків
}
```

Тут виконання функції потоку розбито на етапи. Кожен етап не може бути виконаний без цілковитого завершення попереднього. Створюють *n* потоків, кожен з яких приходить до бар'єра наприкінці етапу. Після переходу бар'єра потоки починають виконувати дії наступного етапу, в кінці якого стоїть новий бар'єр і т. д. Після кожного етапу один із потоків підбиває його результати, викликаючи функцію done() для цього етапу.

Реалізація бар'єрів у POSIX

Ситуація із реалізацією бар'єрів схожа на ситуацію із блокуваннями читання-записування. Бар'єри можуть бути доволі просто реалізовані з використанням м'ютексів та умовних змінних (приклад такої реалізації наведено на сайті підтримки), але зараз POSIX і, приміром, NPTL пропонують їхню підтримку як базових примітивів, що відповідають нашому опису і можуть бути використані безпосередньо.

Для роботи із бар'єрами використовують тип pthread_barrier_t. Ініціалізацію бар'єра здійснюють за допомогою функції pthread_barrier_init(). За параметри вона приймає покажчик на структуру даних бар'єра і кількість потоків, яких можна очікувати на цьому бар'єрі:

```
pthread_barrier_t barrier;
pthread_barrier_init(&barrier, 10);
```

Очікування на бар'єрі відбувається у функції pthread_barrier_wait():

```
int result = pthread_barrier_wait(&barrier);
```

Внаслідок переходу бар'єра одному потокові повертають спеціальне значення PTHREAD_BARRIER_SERIAL_THREAD, усім іншим — нуль.

Якщо бар'єр не потрібний або треба змінити кількість потоків, слід спочатку вилучити об'єкт із пам'яті та звільнити всі його ресурси. Для цього використовують функцію pthread_barrier_destroy():

```
pthread_barrier_destroy(&barrier);
```

Ми ще повернемося до бар'єрів у розділі 7 під час розгляду підходу ведучого-веденого.

5.4. Взаємодія потоків у Linux

Бібліотеки підтримки потоків Linux (LinuxThreads і NPTL) надають програмам користувача набір синхронізаційних примітивів, визначених стандартом POSIX (м'ютекси, умовні змінні, семафори, блокування читання-записування, бар'єри).

Крім того, у NPTL допускають використання цих примітивів у поєднанні з відображуваною або розподілюваною пам'яттю для реалізації міжпроцесової синхронізації.

Описані примітиви реалізовані на основі базових механізмів синхронізації, доступних через системні виклики (у ядрі версії 2.6 до них належать ф'ютекси). Крім того, код ядра може використовувати спеціальні механізми синхронізації, доступні тільки у ядрі.

Далі розглянемо базові механізми синхронізації ядра і застосувань користувача.

5.4.1. Механізми синхронізації ядра Linux

Ядро Linux є *реентерабельним*. Це означає, що одночасно в режимі ядра може виконуватися код кількох процесів. В однопроцесорних системах процесор у конкретний момент виконує код тільки одного процесу, інші перебувають у стані очікування. У багатопроцесорних системах код різних процесів може виконуватися паралельно. Для досягнення реентерабельності всередині ядра повинна бути реалізована така сама синхронізація, що і між потоками одного процесу. Для цього у ядрі передбачені механізми взаємного виключення, які забезпечують безпечний доступ до спільно використовуваних даних ядра.

Аналогом потоків у ядрі виступають *шляхи передачі керування ядра* (kernel control paths). Таким шляхом є послідовність інструкцій, виконуваних ядром для реалізації реакції на системний виклик або обробку переривання. Така послідовність звичайно зводиться до виконання кількох функцій ядра. Наприклад, для обробки системного виклику шлях передачі керування починають з виклику функції `system_call()` і завершують викликом `ret_system_call()`. Надалі іноді говоритимемо не про шляхи керування, а про *процеси в режимі ядра*, маючи на увазі шляхи передачі керування, що відповідають системним викликам, виконаним процесами.

Витісняльність ядра

До останнього часу ядро Linux належало до категорії невитісняльних (nonpreemptive). Це означало, що процес, виконуваний в режимі ядра, не міг бути призупинений (витиснений іншим процесом), поки він сам не вирішить віддати керування. У разі переривання керування після виклику обробника мало бути повернуте в той самий процес.

У ядрі версії 2.6 [90] ситуація змінилася. Це перше ядро, що є витісняльним (preemptive). Тепер процес, виконуваний в режимі ядра, може бути призупинений, коли минув квант часу або почав виконуватися процес із вищим пріоритетом. Після обробки переривання керування теж може бути передане іншому процесові. У результаті скоротився час відгуку системи. Тепер процеси, які проводять занадто багато часу в режимі ядра, не затримуватимуть виконання інших процесів. Природно, що у ядрі все одно залишаються місця, де його не можна витісняти, але тепер їх потрібно виділяти явно.

Необхідність синхронізації у ядрі

Спільно використовувані дані у ядрі можуть бути змінені:

- у коді, викликаному асинхронно внаслідок переривання (до такого коду належать і сам обробник, і *код відкладеної реакції на переривання* (softirq), який пов'язують із обробником, але виконують трохи пізніше);

- у коді, виконуваному на іншому процесорі;
- у коді, що витиснув розглядуваний (у разі витісняльного ядра).

Для забезпечення коректної роботи потрібно завжди забезпечувати синхронізацію доступу до цих структур. Розглянемо механізми такої синхронізації.

Заборона переривань

Для однопроцесорних систем у ядрі можлива проста синхронізація через заборону переривань на час виконання критичних секцій. Даний підхід може бути ефективним тільки тоді, коли критична секція невелика.

Атомарні операції

Як елементарну альтернативу блокуванням ядро Linux пропонує набір *атомарних операцій*. До них належить набір найпростіших операцій (збільшення і зменшення на одиницю, побітові операції), що їх атомарне виконання гарантує ядро. Зазначимо, що саме на базі цих операцій реалізовані складніші примітиви синхронізації, такі як семафори ядра або блокування читання-записування, а також ф'ютекси, на яких ґрунтується реалізація синхронізаційних примітивів режиму користувача.

Спін-блокування

Спін-блокування (spinlocks) — це найпростіші можливі блокування ядра. Вони працюють аналогічно до традиційних м'ютексів (див. розділ 5.3.2), за винятком того, що коли процес у режимі ядра запросить спін-блокування, зайняте у цей час іншим процесом, він не призупиниться, а виконуватиме цикл активного очікування доти, поки блокування не звільниться. У результаті не затрачаються ресурси на призупинення процесу. З іншого боку, такий процес витрачатиме процесорний час, тому спін-блокування краще зберігати недовго. Спін-блокування не є рекурсивними, повторна спроба запровадити таке блокування призводить до взаємного блокування.

Використання спін-блокувань дає змогу вирішити базові проблеми організації взаємного виключення. Для складніших задач можна застосовувати *семафори ядра*.

Семафори ядра

Семафори ядра, на відміну від спін-блокувань, змушують процеси не переходити до активного очікування, а призупинятися, тому їх звичайно використовують тоді, коли очікування може тривати довго (якщо це не так, спін-блокування ефективніші). Семафор ядра за принципом дії не відрізняється від традиційного семафора (див. розділ 5.3.1), він реалізований як структура, що містить цілочисловий лічильник і покажчик на чергу очікування. Структури даних призупинених процесів додають у цю чергу.

Блокування читання-записування

Ядро Linux пропонує також блокування читання-записування, подібні до описаних у розділі 5.3.4. Основна їхня відмінність від традиційної реалізації — режими доступу для читання і для записування повністю розділені (відкриттю для читання відповідає закриття для читання, а відкриттю для записування — закриття для записування). Є також варіант семафора, що розрізняє доступ для читання і для записування.

5.4.2. Синхронізація процесів користувача у Linux. Ф'ютекси

Для того щоб можна було використовувати у застосуваннях різні примітиви синхронізації (м'ютекси, семафори, умовні змінні), на рівні ОС необхідно розробити деяку структуру даних для сигналізації (наприклад, яка відображає лічильник монітора) і забезпечити її спільне використання кількома потоками. Також треба забезпечити можливість переведення потоків у стан очікування, і поновлення одного або одночасно кількох потоків (негайно або через деякий проміжок часу).

Були запропоновані різні способи розв'язання цієї проблеми.

- Підтримка спеціальних засобів синхронізації – *семафорів System V*, які спочатку розробляли для систем із реалізацією моделі процесів. Уся робота з ними заснована на спеціальних структурах даних ядра. Доступ до них здійснюють за допомогою системних викликів. Такий підхід є неефективним, оскільки кожний системний виклик спричиняє виконання кількох сотень машинних інструкцій і перемикань між режимами процесора.
- Використання для операції очікування аналогів системного виклику `sleep()` і реалізація операції поновлення потоків на основі сигналів. Цей підхід було прийнято у бібліотеці LinuxThreads. Застосування сигналів знову передбачало використання системних викликів, а виконання `sleep()` призводило до втрат часу на перемикання контексту.

На практиці найкращим є вирішення, яке використовує системні виклики тільки в разі гострої необхідності й обходиться без перемикання контексту. Вирішення, що відповідає цим вимогам, реалізує швидке блокування режиму користувача (fast user-level locking).

Розглянемо реалізацію такого блокування для двох випадків.

- Потік успішно займає вільне блокування. За звичайного навантаження така ситуація трапляється найчастіше, тому саме для цього випадку потрібно домагатися максимальної ефективності, прагнучи до того, щоб обійтися без системних викликів.
- Є потік, що вже зайняв це блокування. У цьому разі можна виконати системний виклик, щоб призупинити поточний потік. Така ситуація виникає рідше, і втрати продуктивності будуть менші.

Щоб потоки не зверталися до ядра в разі успішного зайняття блокування, вони мають спільно використовувати дані в пам'яті, доступній у режимі користувача. Для потоків одного процесу забезпечити спільне використання нескладно, оскільки в них загальний адресний простір. Для потоків різних процесів потрібно організовувати розподілювану або відображувану пам'ять, докладніше про це буде сказано в розділі 6. Ці спільно використовувані дані називають *блокуванням користувача* (user lock). Вони визначають, закрите чи відкрите блокування і чи є потоки, що очікують на ньому. Потоки атомарно змінюють ці дані у разі спроби заблокування, якщо при цьому виникає необхідність заблокувати або поновити потік, виконують системний виклик, коли такої необхідності немає – потік продовжує виконання в режимі користувача.

Реалізація такого блокування була інтегрована у ядро Linux версії 2.6. Вона дістала назву *ф'ютекс* (futex, від fast user-level mutex – швидкий м'ютекс користувача) [73].

Ф'ютекс — цілочисловий лічильник, що перебуває у спільній пам'яті, яку використовують потоки або процеси. Роботу з цим лічильником ведуть аналогічно до роботи із семафором. Для його збільшення або зменшення потоки мають виконувати атомарні інструкції процесора.

Зазначимо, що лічильник ф'ютекса може розміщатися у спільній пам'яті де завгодно. Потоки можуть перетворити будь-яке місце пам'яті у ф'ютекс без попередньої підготовки (поки не виникне суперництво за ф'ютекс, потоки лише збільшують і зменшують цілочислове значення у спільній пам'яті). Ф'ютексів можна створювати безліч.

Розглянемо ситуації, коли використання ф'ютекса потребує виконання системного виклику.

- У разі спроби заблокуватися на ф'ютексі (коли потрібно його зменшити, а він дорівнює нулю) виконують системний виклик для організації очікування відповідно до операції `futex_wait` (серед інших параметрів йому потрібно передати адресу пам'яті, де перебуває лічильник, і, в окремих випадках, максимальний час очікування). У коді цього виклику адресу пам'яті додають у ядрі до хеш-таблиці й створюють чергу очікування, куди поміщають відповідний керуючий блок потоку. Значення ф'ютекса покладають рівним –1.
- Коли ми збільшуємо ф'ютекс, може з'ясуватися, що вихідним значенням буде –1. У цьому разі виконують системний виклик для поновлення потоків відповідно до операції `futex_wake` (серед параметрів йому, крім адреси пам'яті, треба передати кількість потоків, які потрібно поновити). У результаті потоки переводять із черги очікування в чергу готових процесів.

Легко побачити, що ф'ютекси автоматично роблять можливою синхронізацію потоків різних процесів через розподілювану пам'ять.

Інтерфейс ф'ютексів не призначений для безпосереднього використання у прикладних програмах. Замість цього на основі ф'ютексів бібліотеки підтримки потоків можуть бути реалізовані примітиви синхронізації більш високого рівня (саме це й зроблено у NPTL).

5.5. Взаємодія потоків у Windows XP

5.5.1. Механізми синхронізації потоків ОС

Механізми синхронізації потоків у Windows XP реалізовані на трьох рівнях: синхронізація в ядрі (на рівні потоків ядра), виконавчій системі та в режимі користувача у підсистемі Win32.

Синхронізація в ядрі

Основними механізмами синхронізації ядра Windows XP є спін-блокування. Перед тим як почати роботу зі спільно використовуваними даними, потік ядра має отримати таке блокування, можливо, шляхом активного очікування. Зазначимо, що для однопроцесорних систем у разі одержання спін-блокування замість активного очікування тимчасово маскують переривання від тих джерел, котрі можуть потенційно мати доступ до спільно використовуваних даних.

Синхронізація у виконавчій системі

Код виконавчої системи теж може користуватися спін-блокуваннями, але вони пов'язані з активним очікуванням і їхнє застосування обмежується лише організацією взаємного виключення впродовж короткого часу. Складніші задачі синхронізації розв'язують за допомогою *диспетчерських об'єктів* (dispatcher objects) і *ресурсів виконавчої системи* (executive resources).

Диспетчерські об'єкти — це об'єкти виконавчої системи, що можуть бути використані разом із сервісами очікування менеджера об'єктів.

Ресурси виконавчої системи можуть бути використані тільки в коді ВС, коду користувача вони недоступні. Вони не є об'єктами виконавчої системи, а реалізовані як структури даних зі спеціальними операціями, які можна виконувати над ними. Такий ресурс допускає різні режими доступу — взаємне виключення (як м'ютекс) і доступ за принципом блокування читання-записування.

Об'єкти синхронізації режиму користувача

На основі диспетчерських об'єктів підсистема Win32 реалізує набір об'єктів синхронізації режиму користувача. До них належать, зокрема, м'ютекси, семафори і події. Використання цих об'єктів у прикладних програмах буде розглянуте у розділі 5.5.2.

Очікування на диспетчерських об'єктах

Потік може синхронізуватися з диспетчерським об'єктом через виконання очікування на його дескрипторі. Для цього менеджер об'єктів надає спеціальні сервіси очікування: очікування одного об'єкта і очікування кількох об'єктів (на базі цих сервісів у Win32 API реалізовані функції `WaitForSingleObject()` і `WaitForMultipleObjects()`).

Звертаючись до такого сервісу, потрібно задати дескриптор (або масив дескрипторів) об'єкта, на якому треба очікувати. Після звертання до сервісу очікування потік змінює диспетчерський стан на очікування, після чого ядро вилучає його керуючий блок із черги готових потоків і додає цей блок до черги очікування, пов'язаної з диспетчерським об'єктом.

Диспетчерський об'єкт може перебувати в одному з двох станів: *сигналізованому* (signaled) і *несигналізованому* (nonsignaled). Потік поновлює своє виконання, коли об'єкт, на якому він очікує, переходить у сигналізований стан (відбувається його *сигналізація*). При цьому виконавча підсистема перевіряє чергу очікування цього об'єкта і поновлює виконання одного або кількох потоків з неї (виконуючи перепланування і, можливо, тимчасово підвищуючи їхній пріоритет).

Способи сигналізації та реакція на неї розрізняються для різних об'єктів. Сигналізація зазвичай відбувається явно (коли інший потік викликає спеціальний *метод сигналізації*), однак для деяких об'єктів (процесів або потоків) сигналізація може бути і неявною. Наприклад, об'єкт-потік перебуває в несигналізованому стані впродовж усього свого життєвого циклу, а в разі завершення переходить у сигналізований стан внаслідок чого можуть бути поновлені потоки, які очікують (або приєднали) його. З іншого боку, об'єкт-процес переходить у сигналізований стан, коли завершується останній його потік.

Структури даних синхронізації

Для відстеження того, який потік очікує на який об'єкт, використовують дві структури даних: *диспетчерський заголовок об'єкта* і *блок очікування потоку*.

Диспетчерський заголовок пов'язаний із диспетчерським об'єктом. Він містить інформацію про тип об'єкта; сигнальний стан; покажчик на список блоків очікування потоків, які очікують на цьому об'єкті.

Блок очікування відображає той потік, що очікує на диспетчерському об'єкті. Він містить покажчик на цей диспетчерський об'єкт; покажчик на блок KTHREAD очікувального потоку; покажчик на наступний блок очікування того самого потоку (якщо потік очікує кілька об'єктів); покажчик на наступний блок очікування, пов'язаний з тим самим об'єктом (якщо цей об'єкт очікують кілька потоків); тип очікування (одного або кількох об'єктів); місце дескриптора цього об'єкта в масиві дескрипторів, переданому сервісу очікування у разі очікування кількох об'єктів.

Отже, виконавча система і ядро у будь-який момент можуть отримати інформацію про всі потоки, що очікують на диспетчерському об'єкті, а також інформацію про всі диспетчерські об'єкти, яких очікує потік.

5.5.2. Програмний інтерфейс взаємодії Win32 API

У цьому розділі буде описано реалізацію базових синхронізаційних примітивів Win32 API, які відповідають м'ютексам і умовним змінним. Крім того, Win32 API підтримує семафори [31, 50].

Функції очікування

Розглянемо засоби синхронізації Win32 API, засновані на використанні об'єктів виконавчої системи із дескрипторами. Для всіх таких об'єктів організація очікування зводиться до викликання потоком керування *функції очікування*, що приймає як параметр дескриптор об'єкта (або масив дескрипторів) і перевіряє, чи не відбулася сигналізація цього об'єкта. Умова сигналізації залежить від об'єкта синхронізації.

У разі невиконання умови сигналізації потік переходить у стан очікування, витрачаючи дуже мало ресурсів (процесорного часу тощо), доти, поки сигналізація все-таки не відбудеться або поки не мине максимальний час очікування, якщо він був заданий.

Коли сигналізація відбулася, потік негайно виходить зі стану очікування (із функції очікування) і продовжує своє виконання. Функція очікування в цьому разі перед виходом може змінити стан об'єкта (наприклад, зайняти блокування).

Для очікування сигналізації одного об'єкта у Win32 використовують уже знайому функцію WaitForSingleObject():

```
DWORD WaitForSingleObject(HANDLE handle, DWORD timeout);
```

Параметр handle визначає дескриптор синхронізаційного об'єкта, а параметр timeout задає максимальний час очікування в мілісекундах (значення INFINITE свідчить про нескінченне очікування). Функція WaitForSingleObject() повертає такі значення: WAIT_OBJECT_0 — відбулася сигналізація об'єкта; WAIT_TIMEOUT — минув час очікування (якщо timeout не дорівнював INFINITE), а об'єкт свого стану так і не змінив.

Можна очікувати сигналізації не одного об'єкта, а кількох одразу (деякий аналог використання кількох умов очікування для умовної змінної). Для цього використовують функцію WaitForMultipleObjects():

```
DWORD WaitForMultipleObjects(
    DWORD count,            // довжина масиву дескрипторів
    CONST HANDLE *handles,  // масив дескрипторів
    BOOL waitall,           // прапорець режиму очікування
    DWORD timeout           // аналогічно до WaitForSingleObject
);
```

Функція приймає масив дескрипторів handles завдовжки count (максимальна довжина масиву 64 елементи). Режим очікування і повернене значення залежать від прапорця waitall.

- Якщо waitall дорівнює TRUE, задано режим очікування всіх об'єктів. Очікування завершується у разі здійснення сигналізації всіх об'єктів, функція поверне WAIT_OBJECT_0.
- Якщо waitall дорівнює FALSE, задано режим очікування одного об'єкта. Очікування завершується у разі здійснення сигналізації хоча б одного з об'єктів, функція поверне WAIT_OBJECT_0+*i*, де *i* — індекс дескриптора цього об'єкта в масиві handles.

М'ютекси Win32 API

Найпростішими синхронізаційними об'єктами із дескрипторами є м'ютекси. Для створення м'ютекса використовують функцію CreateMutex():

```
HANDLE CreateMutex(  // повертає дескриптор м'ютекса
    // атрибути безпеки, нульовий покажчик, якщо за замовчуванням
    LPSECURITY_ATTRIBUTES sec_attrs,
    BOOL init_locked,     // відразу зайнятий поточним потоком, якщо TRUE
    LPCTSTR mutex_name ); // ім'я або нульовий покажчик, якщо не потрібно
```

Ім'я м'ютекса задавати не обов'язково, якщо потрібно звертатися до м'ютекса тільки в рамках цього процесу.

Ось приклад створення м'ютекса:

```
HANDLE mutex = CreateMutex (0, FALSE, 0);
```

Щоб зайняти м'ютекс, необхідно викликати функцію очікування:

```
WaitForSingleObject(mutex, INFINITE);
```

Якщо м'ютекс вільний, потік його займає і продовжує виконання. Коли м'ютекс зайнятий (заблокований іншим потоком), то поточний потік очікує, поки його не буде звільнено (або поки не мине час очікування, якщо другий параметр не дорівнює INFINITE). У першому випадку він блокує м'ютекс, у другому — ні, і WaitForSingleObject()повертає WAIT_TIMEOUT.

Звільнення м'ютекса здійснює функція ReleaseMutex(mutex). При цьому відбувається сигналізація м'ютекса і потік, що його очікує, поновлюється. Якщо кілька потоків очікували один і той самий м'ютекс, його буде зайнято одним із потоків, при цьому не можна заздалегідь передбачити яким. Після використання дескриптор потрібно закрити функцією CloseHandle(mutex).

Наведемо приклад роботи із м'ютексом Win32 API.

```
HANDLE mutex = CreateMutex (0, FALSE, 0);
WaitForSingleObject(mutex, INFINITE);
// критична секція
ReleaseMutex(mutex);
CloseHandle(mutex);
```

М'ютекси Win32 API є рекурсивними.

Альтернативою м'ютексам є *об'єкти критичних секцій*, які не пов'язані з об'єктами виконавчої системи і працюють ефективніше, але можуть бути використані тільки для синхронізації об'єктів одного процесу. Приклад використання об'єктів критичних секцій наведено на сайті супроводу.

Події

Розглянемо засоби, які надає Win32 API для роботи з умовними змінними. Зазначимо відразу, що прямої реалізації цього примітива у Win32 API немає [50, 100]. Спробу реалізувати частину функціональності умовних змінних буде зроблено трохи пізніше, а поки що зупинимося на *подіях* (events) — примітивах, які можна використати для цього.

Події — це засоби сигналізації об'єктів для виклику деяких дій. Відповіддю на події звичайно є поновлення одного або кількох потоків. Фактично події можна розуміти як спробу реалізувати функціональність умовних змінних без інтеграції з м'ютексами. Але, оскільки саме інтеграція із м'ютексами в рамках концепції монітора забезпечує безпеку використання умовних змінних, відмова від такої інтеграції призводить до того, що під час роботи з подіями виникають певні проблеми. Потрібно враховувати можливість «втручання» інших потоків між передачею події та реакцією на неї.

Розрізняють два типи подій — з *автоматичним* (auto reset event) та *ручним скиданням* (manual reset event), а також два засоби їхньої сигналізації — функції `PulseEvent()` і `SetEvent()`. Будь-який спосіб сигналізації можна використати з різним типом подій. У поєднанні це дає чотири варіанти дій. Розглянемо такі дії одну за одною, але спочатку зупинимося на тому, як створювати події.

Для створення події використовують функцію `CreateEvent()`:

```
HANDLE CreateEvent(          // повертає дескриптор створеної події
    LPSECURITY_ATTRIBUTES sec_attrs, // може бути 0
    BOOL manual_reset,       // тип події
    // якщо TRUE – після створення відразу сигналізувати
    BOOL init_state,
    LPCTSTR evt_name );      // рядок з ім'ям або нульовий покажчик
```

Коли значення `manual_reset` дорівнює `TRUE`, то маємо подію із ручним скиданням, а коли воно дорівнює `FALSE` — з автоматичним.

```
HANDLE event = CreateEvent(0, TRUE, FALSE, 0);  // ручне скидання
```

Для очікування сигналізації події використовують функцію очікування

```
WaitForSingleObject(event, INFINITE);
```

Тепер розглянемо різні варіанти зміни стану подій. Насамперед зазначимо, що всі функції сигналізації події та скидання її стану мають однаковий синтаксис:

```
BOOL SetEvent(HANDLE event);      // два види
BOOL PulseEvent(HANDLE event);    // сигналізації подій
BOOL ResetEvent(HANDLE event);    // скидання події
```

Під час виконання функції `SetEvent()` відбувається сигналізація події. Подальші дії залежать від типу події.

- Для подій з автоматичним скиданням поновлюють виконання одного потоку, який очікує на події. Якщо жоден потік на події не очікує, подія залишається в сигналізованому стані доти, поки якийсь потік не спробує почати очікування на цій самій події. Після цього потік негайно продовжує виконання, а стан події автоматично скидається (наступні потоки очікуватимуть).
- Для подій зі ручним скиданням поновлюються всі потоки, що очікують, після чого подія залишається в сигналізованому стані, тому всі наступні потоки негайно продовжуватимуть роботу в разі спроби виконати очікування. Подія залишатиметься в такому стані доти, поки якийсь потік не скине її вручну, викликавши `ResetEvent()`.

Виконання функції `PulseEvent()` подібне до двох послідовних викликів `SetEvent()` і `ResetEvent()`. При цьому подію сигналізує і поновлює один із потоків, які очікують на події (для подій з автоматичним скиданням), або всі потоки, що очікують (для подій з ручним скиданням), затим стан події скидають. Коли жодного потоку, що очікує немає, подію негайно скидають і факт її сигналізації зникає.

Виклик функції `SetEvent()` зберігає стан події, а виклик функції `PulseEvent()` — ні.

Реалізація умовних змінних за допомогою подій

Очевидно, що за допомогою функції `PulseEvent()` можна виконувати дії, аналогічні до сигналізації умовних змінних. Виконання цієї функції для події з автоматичним скиданням аналогічне до операції `signal`, а для події із ручним скиданням — до операції `broadcast`.

На жаль, основна проблема, пов'язана з реалізацією умовних змінних на основі подій, полягає не в сигналізації, а в організації очікування. Насправді коректно реалізувати першу частину `wait` (атомарне звільнення м'ютекса і перехід у стан очікування) на основі стандартних функцій очікування (`WaitForSingleObject()` або `WaitForMultipleObjects()`) неможливо. Якщо спочатку звільняти м'ютекс, а потім починати очікувати подій, отримаємо помилкове поновлення, розглянуте в розділі 5.3.3.

Деяким компромісом тут може бути використання функції `SignalObjectAndWait()`, що вперше з'явилася у Windows NT 4.0 і відтоді доступна у системах лінії Windows XP. Ця функція дає змогу атомарно сигналізувати один об'єкт і почати очікування зміни стану іншого.

```
DWORD SignalObjectAndWait(
    HANDLE object_to_signal, // дескриптор сигналізованого об'єкта
    HANDLE object_to_wait,   // дескриптор об'єкта, на який ми очікуємо
    DWORD timeout,           // максимальний інтервал очікування
    BOOL alertable           // може бути FALSE
);
```

У нашій ситуації така функція може надати можливість просто реалізувати умовні змінні з однією операцією сигналізації — signal або broadcast, чого може бути достатньо для багатьох застосувань. Коли ж потрібна умовна змінна, що одночасно має очікувати обидві операції відразу, можна скористатися складнішими вирішеннями [100].

```
HANDLE cond_signal_init() {
    // подія з автоматичним скиданням
    HANDLE signal_event = CreateEvent(0, FALSE, FALSE, 0);
    return signal_event;
}
HANDLE cond_broadcast_init() {
    // подія з ручним скиданням
    HANDLE broadcast_event = CreateEvent(0, TRUE, FALSE, 0);
    return broadcast_event;
}

// виконання signal
void cond_signal_send (HANDLE signal_event) {
    PulseEvent(signal_event);
}
// виконання broadcast
void cond_broadcast_send (HANDLE broadcast_event) {
    PulseEvent(broadcast_event);
}

// wait для signal
void cond_signal_wait (HANDLE signal_event, HANDLE mutex) {
    // звільняємо м'ютекс і чекаємо signal
    SignalObjectAndWait(mutex, signal_event, INFINITE, FALSE);
    // після приходу події знову займаємо м'ютекс
    WaitForSingleObject(mutex, INFINITE);
}
// wait для broadcast
void cond_broadcast_wait (HANDLE broadcast_event, HANDLE mutex) {
    // звільняємо м'ютекс і чекаємо broadcast
    SignalObjectAndWait(mutex, broadcast_event, INFINITE, FALSE);
    // після приходу події знову займаємо м'ютекс
    WaitForSingleObject(mutex, INFINITE);
}
```

За наявності у Win32 API функції SignalObjectAndWaitMultiple() з очікуванням на зразок WaitForMultipleObjects() можна було б очікувати дві події відразу — signal і broadcast, що повністю розв'язувало б задачу реалізації умовної змінної.

```
HANDLE events[] = { signal_event, broadcast_event };
// такої функції немає, на жаль
SignalObjectAndWaitMultiple(mutex, events, INFINITE, FALSE);
```

Висновки

- Потоки одного процесу, що взаємодіють, мають доступ до спільно використовуваних даних, розміщених в адресному просторі цього процесу. Будь-який потік здатний у будь-який момент часу змінити ці дані й спричинити стан змагання, коли результат залежить від послідовності виконання потоків.

- Для забезпечення коректного доступу до спільно використовуваних даних застосовують механізми синхронізації потоків; основним з них є забезпечення взаємного виключення, коли в конкретний момент часу доступ до таких даних може мати тільки один потік. Для організації взаємного виключення використовують блокування.
- Для розв'язання задач синхронізації можна використовувати різні синхронізаційні примітиви. До найпростіших примітивів належать м'ютекси, семафори, умовні змінні, блокування читання-записування і бар'єри.
- Механізмом більш високого рівня є концепція монітора, що поєднує м'ютекси та умовні змінні, а також задає деякі правила їхньої взаємодії для захисту спільно використовуваних даних. Використання моніторів є найпослідовнішим підходом до синхронізації потоків.

Контрольні запитання та завдання

1. Опишіть проблему, що виникає під час паралельного виконання декількома потоками такого коду:

```
void f() {
    static int i = 0;
    i++;
}
```

2. Внаслідок виконання атомарної інструкції SWAP(r,m) міняються місцями дані в регістрі r і комірці пам'яті m. Наведіть приклад реалізації блокування на основі даної інструкції, подібно до того, як це було реалізовано в розділі 5.2.2 з використанням інструкції TSL.
3. Запропонована нова апаратна конструкція з метою вирішення проблеми взаємного виключення — атомарний лічильник. Значення такого лічильника може бути зчитане і збільшене на одиницю за одну атомарну операцію: i = c++. Крім того, внаслідок атомарної операції лічильник можна обнулити. Покажіть, як можна реалізувати взаємне виключення на основі атомарного лічильника.
4. У чому принципова відмінність активного очікування від інших видів очікування, реалізованих в операційній системі?
5. Проаналізуйте виконання коду трьох потоків, що синхронізуються за допомогою семафорів:

```
semaphore L = 3, R = 0;
void thread1_fun() {    void thread2_fun() {    void thread3_fun() {
   for(; ;) {              for(; ;) {              for(; ;) {
     down(L);                down(R);                down(R);
     printf("C");            printf("A");            printf("D");
     up(R);                  printf("B");
   }                         up(R);                }
                           }
}                       }                       }
```

а) скільки символів D буде відображено після запуску цих потоків?

б) яка мінімальна кількість символів A буде відображена після запуску цих потоків?

в) як ви вважаєте, чи може внаслідок виконання цих потоків відобразитися рядок CABABDDCABCABD?

г) чи може внаслідок виконання вказаних потоків бути відображений рядок CABACDBCABDD?

6. Два потоки паралельно виконують код:

```
Потік 1              Потік 2
а) x = 1;            е) x += 2;
б) lock(mutex);      ж) lock(mutex);
в) y = y + x;        з) y--;
г) x = 2;            і) x = x - y;
д) unlock(mutex);    к) unlock(mutex);
```

Спочатку змінним x та y надано значення 0, і м'ютекс відкритий. Всі інструкції обох потоків є атомарними. Назвіть допустимі пари значень, що можуть бути надані змінним x та y внаслідок виконання цих потоків, і відповідні послідовності кроків виконання потоків.

7. Припустімо, що MESA-монітор модифіковано таким чином, що у разі сигналізації умовної змінної поновлюваний потік переміщують у початок черги готових потоків. У випадку припинення або завершення даного потоку планувальник негайно запускає на виконання поновлюваний потік. Чи означає це, що помилкове поновлення стало неможливим?

8. Чи еквівалентні умовні змінні семафорам, які завжди ініціалізуються нулем і мають таку властивість, що потік, який починає очікування на семафорі, звільняє м'ютекс (у цьому випадку операція `signal` аналогічна up, а `wait` — down зі звільненням м'ютекса)?

9. Напишіть код таких функцій:

а) `send_nmsg()`, що відсилає повідомлення N потокам і припиняє поточний потік, поки усі вони не одержать повідомлення;

б) `recv_msg()`, що припиняє даний потік до одержання відісланого за допомогою `send_nmsg()` повідомлення.

Використовуйте потоки POSIX і Win32. Для потоків Win32 моделюйте умовні змінні з використанням подій.

10. Реалізуйте спільно використовувану динамічну структуру даних (стек, двозв'язний список, бінарне дерево) з використанням потоків POSIX і Win32. Функції доступу до цієї структури даних оформіть, якщо це можливо, у вигляді монітора.

11. Розробіть програму реалізації блокувань читання-записування з перевагою записування. Використайте потоки POSIX. Які труднощі можуть виникнути у разі реалізації таких блокувань для потоків Win32?

Розділ 6
Міжпроцесова взаємодія

- Проблеми міжпроцесової взаємодії
- Види міжпроцесової взаємодії
- Принципи та базові примітиви передавання повідомлень
- Технології передавання повідомлень

Дотепер ми розглядали взаємодію потоків одного процесу. Головною особливістю цієї взаємодії є простота технічної реалізації обміну даними між ними — усі потоки одного процесу використовують один адресний простір, а отже, можуть вільно отримувати доступ до спільно використовуваних даних, ніби вони є їх власними. Оскільки технічних труднощів із реалізацією обміну даними тут немає, основною проблемою, яку потрібно вирішувати в цьому випадку, є синхронізація потоків.

З іншого боку, кожен потік виконується в рамках адресного простору деякого процесу, тому часто постає задача організації взаємодії між потоками різних процесів. Ідеться власне про міжпроцесову взаємодію (interprocess communication, IPC) [37]. Ця технологія з'явилася задовго до поширення багатопотоковості.

Для потоків різних процесів питання забезпечення синхронізації теж є актуальними, але вони в більшості випадків не ґрунтуються на понятті спільно використовуваних даних (такі дані за замовчуванням для процесів відсутні). Крім того, додається досить складна задача забезпечення обміну даними між захищеними адресними просторами. Підходи до її розв'язання визначають різні види міжпроцесової взаємодії.

6.1. Види міжпроцесової взаємодії

Реалізація міжпроцесової взаємодії здійснюється трьома основними методами: *передавання повідомлень*, *розподілюваної пам'яті* та *відображуваної пам'яті*. Ще одним методом IPC також можна вважати технологію сигналів, що була розглянута в розділі 3.

Сигнали були розглянуті раніше, тому що їхнє використання не зводиться тільки до організації IPC (синхронні сигнали є засобом оповіщення процесу про виняткову ситуацію); без них складно пояснити ряд базових понять керування процесами (наприклад, очікування завершення процесу).

6.1.1. Методи розподілюваної пам'яті

Методи розподілюваної пам'яті (shared memory) дають змогу двом процесам обмінюватися даними через загальний буфер пам'яті. Перед обміном даними кожний із тих процесів має приєднати цей буфер до свого адресного простору з використанням спеціальних системних викликів (перед цим перевіряють права). Буфер доступний у системі за допомогою процедури іменування, термін його існування звичайно обмежений часом роботи всієї системи. Дані в ньому фактично є спільно використовуваними, як і для потоків. Жодних засобів синхронізації доступу до цих даних розподілювана пам'ять не забезпечує, програміст, так само, як і при розробці багатопотокових застосувань, має організовувати її сам.

6.1.2. Методи передавання повідомлень

В основі методів передавання повідомлень (message passing) лежать різні технології, що дають змогу потокам різних процесів (які, можливо, виконуються на різних комп'ютерах) обмінюватися інформацією у вигляді фрагментів даних фіксованої чи змінної довжини, котрі називають *повідомленнями* (messages). Процеси можуть приймати і відсилати повідомлення, при цьому автоматично забезпечується їхнє пересилання між адресними просторами процесів одного комп'ютера або через мережу. Важливою особливістю технологій передавання повідомлень є те, що вони не спираються на спільно використовувані дані — процеси можуть обмінюватися повідомленнями, навіть не знаючи один про одного.

6.1.3. Технологія відображуваної пам'яті

Ще однією категорією засобів міжпроцесової взаємодії є *відображувана пам'ять* (mapped memory). У ряді ОС відображувана пам'ять є базовим системним механізмом, на якому ґрунтуються інші види міжпроцесової взаємодії та системні вирішення. Звичайно відображувану пам'ять використовують у поєднанні з інтерфейсом файлової системи, в такому разі говорять про *файли, відображувані у пам'ять* (memory-mapped files).

Ця технологія зводиться до того, що за допомогою спеціального системного виклику (зазвичай це `mmap()`) певну частину адресного простору процесу однозначно пов'язують із вмістом файла. Після цього будь-яка операція записування в таку пам'ять спричиняє зміну вмісту відображеного файла, яка відразу стає доступною усім застосуванням, що мають доступ до цього файла. Інші застосування теж можуть відобразити той самий файл у свій адресний простір і обмінюватися через нього даними один з одним.

Файли, відображувані у пам'ять, будуть докладно розглянуті в розділі 11.

6.1.4. Особливості міжпроцесової взаємодії

Тепер можна порівняти характеристики міжпроцесової взаємодії із характеристиками взаємодії потоків одного процесу.

- Проблема організації обміну даними є актуальною тільки для міжпроцесової взаємодії, оскільки потоки обмінюються даними через загальний адресний простір. Обмін даними між потоками схожий на використання розподілюваної пам'яті, але не потребує підготовчих дій.

- Проблема синхронізації доступу до спільно використовуваних даних є актуальною для взаємодії потоків і для міжпроцесової взаємодії із використанням розподілюваної пам'яті. Використання механізму передавання повідомлень не ґрунтується на спільно використовуваних даних.

6.2. Базові механізми міжпроцесової взаємодії

У цьому розділі розглянемо особливості організації взаємодії між потоками *різних процесів*. Основною характеристикою такої взаємодії є те, що у процесів немає спільного адресного простору, тому тут не можна безпосередньо працювати зі спільно використовуваними даними, як це було можливо для потоків. Тут ітиметься переважно про *процеси*, під якими розуміють потоки різних процесів.

6.2.1. Міжпроцесова взаємодія на базі спільної пам'яті

Для вирішення проблеми міжпроцесової синхронізації необхідно:

- по-перше, організувати спільну пам'ять між процесами (це може бути розподілювана пам'ять або файл, відображений у пам'ять);
- по-друге, розмістити в цій пам'яті стандартні синхронізаційні об'єкти (семафори, м'ютекси, умовні змінні);
- по-третє, використовуючи ці об'єкти, працювати зі спільно використовуваними даними, як це робилося у разі використання потоків.

Такий підхід широко застосовують на практиці. На жаль, досить складно запропонувати спосіб його реалізації для міжпроцесової синхронізації у більшості систем, оскільки різні системи пропонують різний набір засобів організації спільної пам'яті та засобів сигналізації, які можуть працювати в такій пам'яті. Універсальним рішенням у даному разі є застосування семафорів.

6.2.2. Основи передавання повідомлень

Усі методи взаємодії, які було розглянуто дотепер, ґрунтуються на читанні й записуванні спільно використовуваних даних. На практиці така взаємодія не завжди можлива (наприклад, робота зі спільно використовуваними даними проблематична, якщо для процесів немає спільної фізичної пам'яті, а є тільки мережний зв'язок між комп'ютерами, на яких вони виконуються). У таких випадках можна використати засоби взаємодії, які не ґрунтуються на спільно використовуваних даних, передусім *засоби передавання повідомлень* [27, 37, 57].

Як було вже згадано, засоби передавання повідомлень ґрунтуються на обміні повідомленнями — фрагментами даних змінної довжини. Основою такого обміну є не спільна пам'ять, а *канал зв'язку* (communication channel). Він забезпечує взаємодію між процесами (для того, щоб спілкуватися, вони повинні створити канал зв'язку) і є абстрактним відображенням мережі зв'язку. Абстрактність каналу дає змогу реалізувати його не тільки на основі мережної взаємодії, але й спільної пам'яті (коли процеси перебувають на одному комп'ютері). При цьому такі зміни в реалізації будуть сховані від процесів, що взаємодіють.

Виокремлюють такі характеристики каналів зв'язку: спосіб задання; кількість
ощесів, які можуть бути з'єднані одним каналом; кількість каналів, *які можуть*
ти створені між двома процесами; пропускна здатність каналу (кількість пові-
лень, які можуть одночасно перебувати в системі й бути асоційованими з цим
алом); максимальний розмір повідомлення; спрямованість зв'язку через ка-
л (двобічний або однобічний зв'язок).

В однобічному зв'язку для конкретного процесу допускають передавання да-
х тільки в один бік.

имітиви передавання повідомлень

новна особливість передавання повідомлень полягає в тому, що процеси спіль-
використовують тільки канали. Немає необхідності забезпечувати взаємне ви-
очення процесів під час доступу до спільно використовуваних даних, замість
ого досить визначити *примітиви передавання повідомлень* — спеціальні опера-
обміну даними через канал, які забезпечують не лише обмін даними, але й син-
онізацію.

Є два примітиви передавання повідомлень: send (для відсилання повідомлен-
каналом) і receive (для отримання повідомлення з каналу).

Розглянемо, як особливості реалізації send і receive дають змогу виділити різ-
класи методів передавання повідомлень.

Зазначені примітиви передавання повідомлень можуть задавати *прямий* і *не-
мий* обмін даними. При прямому обміні даними необхідно явно вказувати
ощес, з яким необхідно обмінюватись інформацією. Непрямий обмін здійснюють
ез спеціальний об'єкт (поштову скриньку, порт); процеси можуть поміщати
відомлення в поштову скриньку і отримувати їх звідти. Зазвичай кілька процесів
оть доступ до однієї поштової скриньки, застосовуючи під час її пошуку мето-
іменування. Більшість сучасних технологій обміну повідомленнями використо-
: непрямий обмін даними. Прикладом прямого обміну є традиційні сигнали.

нхронне й асинхронне передавання повідомлень

инимося на основних питаннях синхронізації під час передавання повідомлень.
жна виокремити різні групи методів передавання повідомлень залежно від то-
як вони дають можливість відповісти на два запитання.

Чи може потік бути призупинений під час виконання операції send, якщо пові-
домлення не було отримане?

Чи може потік бути призупинений під час виконання операції receive, якщо
повідомлення не було відіслане?

У реальних системах відповідь на друге запитання практично завжди буде по-
ивною — неблокувальне приймання повідомлень спричиняється до того, що
ни губляться. Варіанти відповідей на перше запитання визначають два класи
едавання повідомлень — *синхронне* і *асинхронне*.

Під час синхронного передавання повідомлень операція send призупиняє про-
: до отримання повідомлення, а під час асинхронного передавання повідомлень
а не призупиняє процес (тобто є *неблокувальною*); після відсилання повідомлення
ощес продовжує своє виконання, не чекаючи отримання результату. Найзручні-
в цьому випадку використати непряму адресацію через поштові скриньки.

Реалізація синхронного й асинхронного передавання повідомлень залежить від низки характеристик каналу й обміну повідомленнями, насамперед від пропускної здатності каналу.

- Якщо пропускна здатність дорівнює нулю (повідомлення не можуть очікувати в системі), відправник завжди має очікувати, поки одержувачу не надійде повідомлення, а одержувач має очікувати, поки повідомлення йому не буде відіслано. Два процеси мають явно домовлятися про майбутній обмін.
- Якщо пропускна здатність обмежена (у системі можуть перебувати максимум n повідомлень для цього каналу), відправник має очікувати тільки тоді, коли черга повідомлень для цього каналу переповнена (у ній перебуває рівно n повідомлень), одержувач має очікувати, якщо ця черга порожня.
- Якщо пропускна здатність необмежена, очікування можливе тільки для одержувача за порожньої черги.

Під час обміну повідомленнями необхідне підтвердження їх отримання. Деякі методи обміну повідомленнями не вимагають підтвердження зовсім, в інших випадках можлива ситуація, коли відправника після виконання операції send блокують доти, поки одержувач не надішле йому інше повідомлення із підтвердженням отримання; таку технологію називають *обміном повідомленнями із підтвердженням отримання*.

Розв'язання задачі виробників-споживачів за допомогою передавання повідомлень

Розглянемо розв'язання задачі виробників-споживачів із використанням асинхронного передавання повідомлень. Організовують дві поштові скриньки — для виробника і для споживача. Якщо скринька порожня, потік очікуватиме, поки в ній не з'явиться повідомлення.

Поштові скриньки виробника та споживача мають свої особливості.

Скринька виробника може містити тільки порожні повідомлення загальною кількістю не більше n. Наявність m повідомлень у цій скриньці служить сигналом для виробника, що в буфері є місце для m об'єктів. Щоб відіслати дані в буфер, виробник забирає зі скриньки одне повідомлення, заповнює його даними і відсилає в скриньку споживача. Заповнивши буфер, виробник спустошить свою скриньку і буде змушений чекати, поки споживач не помістить у неї порожнє повідомлення.

Повідомлення у скриньці споживача відповідають об'єктам у буфері. Після того як споживач забере повідомлення з цієї скриньки, він використає його дані й відішле порожнє повідомлення у скриньку виробника, сигналізуючи, що в буфері з'явилося місце. Порожня скринька споживача означає порожній буфер — споживач чекатиме, поки виробник не помістить заповнене повідомлення в цю скриньку.

На початку роботи скриньку виробника заповнюють порожніми повідомленнями загальним числом n (це буде означати, що він може зробити n об'єктів).

Функції `producer()` і `consumer()` схожі. І виробник, і споживач у циклі намагаються забрати повідомлення зі своїх скриньок. Якщо це вдається виробникові, він заповнює повідомлення даними і відсилає його у скриньку споживача, якщо це зможе зробити споживач, він скористається повідомленням і відішле порожнє повідомлення у скриньку виробника. Після цього цикл триває.

Ось алгоритм розв'язання задачі:

```
message_t null_msg;     // порожнє повідомлення
mailbox_t producer_mailbox, consumer_mailbox;   // поштові скриньки
int n = 100;            // розмір буфера
void producer() {
    message_t producer_msg;
    for (; ;) {
        // забрати повідомлення зі скриньки виробника,
        // чекати, якщо вона порожня
        receive(producer_mailbox, producer_msg);
        producer_msg.data = produce();
        // відіслати заповнене повідомлення у скриньку споживача
        send(consumer_mailbox, producer_msg);
    }
}
void consumer() {
    message_t consumer_msg;
    for (; ;) {
        // забрати повідомлення зі скриньки споживача, чекати якщо вона порожня
        receive(consumer_mailbox, consumer_msg);
        consume(consumer_msg.data);
        // відіслати у скриньку виробника порожнє повідомлення
        // повідомивши йому про те, що у буфері є місце
        send(producer_mailbox, null_msg);
    }
}
void main() {
    // заповнити скриньку виробника порожніми повідомленнями
    for(int i=0; i<n; i++)
        send(producer_mailbox, null_msg);
    // запустити producer() і customer() паралельно
}
```

Основна відмінність цього розв'язання від запропонованих у розділі 5 полягає в тому, що воно не залежить від спільно використовуваних даних. Доступ до поштових скриньок може бути виконаний за допомогою системних викликів, що приховують їхнє місцезнаходження; скриньки можуть бути й віддаленими. Це дає змогу застосовувати алгоритм тоді, коли виробник і споживач є різними процесами, а можливо, й перебувають на різних комп'ютерах.

6.2.3. Технології передавання повідомлень

Розглянемо методи передавання повідомлень, які застосовують на практиці.

Канали

Канал — це найпростіший засіб передавання повідомлень. Він є циклічним буфером, записування у який виконують за допомогою одного процесу, а читання — за допомогою іншого. У конкретний момент часу до каналу має доступ тільки один процес. Операційна система забезпечує синхронізацію згідно правилу: якщо процес намагається записувати в канал, у якому немає місця, або намагається зчитати більше даних, ніж поміщено в канал, він переходить у стан очікування.

Розрізняють безіменні та поіменовані канали.

До *безіменних каналів* немає доступу за допомогою засобів іменування, тому процес не може відкрити вже наявний безіменний канал без його дескриптора. Це означає, що такий процес має отримати дескриптор каналу від процесу, що його створив, а це можливо тільки для зв'язаних процесів.

До *поіменованих каналів* (named pipes) є доступ за іменем. Такому каналу може відповідати, наприклад, файл у файловій системі, при цьому будь-який процес, який має доступ до цього файла, може обмінюватися даними через відповідний канал. Поіменовані канали реалізують непрямий обмін даними.

Обмін даними через канал може бути однобічним і двобічним.

Приклади використання поіменованих каналів будуть наведені в розділі 11, безіменних — у розділі 17.

Черги повідомлень

Іншою технологією асинхронного непрямого обміну даними є застосування *черг повідомлень* (message queues) [37, 52]. Для таких черг виділяють спеціальне місце в системній ділянці пам'яті ОС, доступне для застосувань користувача. Процеси можуть створювати нові черги, відсилати повідомлення в конкретну чергу й отримувати їх звідти. Із чергою одночасно може працювати кілька процесів. Повідомлення — це структури даних змінної довжини. Для того щоб процеси могли розрізняти адресовані їм повідомлення, кожному з них присвоюють тип. Відіслане повідомлення залишається в черзі доти, поки не буде зчитане. Синхронізація під час роботи з чергами схожа на синхронізацію для каналів.

Сокети

Найрозповсюдженішим методом обміну повідомленнями є використання *сокетів* (sockets). Ця технологія насамперед призначена для організації мережного обміну даними, але може бути використана й для взаємодії між процесами на одному комп'ютері (власне, мережну взаємодію можна розуміти як узагальнення IPC).

Сокет — це абстрактна кінцева точка з'єднання, через яку процес може відсилати або отримувати повідомлення. Обмін даними між двома процесами здійснюють через пару сокетів, по одному на кожен процес. Абстрактність сокету полягає в тому, що він приховує особливості реалізації передавання повідомлень — після того як сокет створений, робота з ним не залежить від технології передавання даних, тому один і той самий код можна без великих змін використовувати для роботи із різними протоколами зв'язку.

Особливості протоколу передавання даних і формування адреси сокету визначає *комунікаційний домен;* його потрібно зазначати під час створення кожного сокету. Прикладами доменів можуть бути *домен Інтернету* (який задає протокол зв'язку на базі TCP/IP) і *локальний домен* або *домен UNIX*, що реалізує зв'язок із використанням імені файла (подібно до поіменованого каналу). Сокет можна використовувати у поєднанні тільки з одним комунікаційним доменом. *Адреса сокету* залежить від домену (наприклад, для сокетів домену UNIX такою адресою буде ім'я файла).

Способи передавання даних через сокет визначаються його *типом*. У конкретному домені можуть підтримуватися або не підтримуватися різні типи сокетів.

Наприклад, і для домену Інтернет, і для домену UNIX підтримуються сокети таких типів:

- *потокові* (stream sockets) – задають надійний двобічний обмін даними суцільним потоком без виділення меж (операція читання даних повертає стільки даних, скільки запитано або скільки було на цей момент передано);
- *дейтаграмні* (datagram sockets) – задають ненадійний двобічний обмін повідомленнями із виділенням меж (операція читання даних повертає розмір того повідомлення, яке було відіслано).

Під час обміну даними із використанням сокетів зазвичай застосовується технологія клієнт-сервер, коли один процес (сервер) очікує з'єднання, а інший (клієнт) з'єднують із ним.

Перед тим як почати працювати з сокетами, будь-який процес (і клієнт, і сервер) має створити сокет за допомогою системного виклику `socket()`. Параметрами цього виклику задають комунікаційний домен і тип сокету. Цей виклик повертає *дескриптор сокету* – унікальне значення, за яким можна буде звертатися до цього сокету.

Подальші дії відрізняються для сервера і клієнта. Спочатку розглянемо послідовність кроків, яку потрібно виконати для сервера.

1. Сокет пов'язують з адресою за допомогою системного виклику `bind()`. Для сокетів домену UNIX як адресу задають ім'я файла, для сокетів домену Інтернету – необхідні характеристики мережного з'єднання. Далі клієнт для встановлення з'єднання й обміну повідомленнями має буде вказати цю адресу.
2. Сервер дає змогу клієнтам встановлювати з'єднання, виконавши системний виклик `listen()` для дескриптора сокету, створеного раніше.
3. Після виходу із системного виклику `listen()` сервер готовий приймати від клієнтів запити на з'єднання. Ці запити вишиковуються в чергу. Для отримання запиту із цієї черги і створення з'єднання використовують системний виклик `accept()`. Внаслідок його виконання в застосування повертають новий сокет для обміну даними із клієнтом. Старий сокет можна використовувати далі для приймання нових запитів на з'єднання. Якщо під час виклику `accept()` запити на з'єднання в черзі відсутні, сервер переходить у стан очікування.

Для клієнта послідовність дій після створення сокету зовсім інша. Замість трьох кроків досить виконати один – встановити з'єднання із використанням системного виклику `connect()`. Параметрами цього виклику задають дескриптор створеного раніше сокету, а також адресу, подібну до вказаної на сервері для виклику `bind()`.

Після встановлення з'єднання (і на клієнті, і на сервері) з'явиться можливість передавати і приймати дані з використанням цього з'єднання. Для передавання даних застосовують системний виклик `send()`, а для приймання – `recv()`.

Зазначену послідовність кроків використовують для встановлення надійного з'єднання. Якщо все, що нам потрібно, – це відіслати і прийняти конкретне повідомлення фіксованої довжини, то з'єднання можна й не створювати зовсім. Для цього як відправник, так і одержувач повідомлення мають попередньо зв'язати сокети з адресами через виклик `bind()`. Потім можна скористатися викликами прямого передавання даних: `sendto()` – для відправника і `recvfrom()` – для одер-

жувача. Параметрами цих викликів задають адреси одержувача і відправника, а також адреси буферів для даних.

Докладніше використання сокетів буде описано в розділі 16.

Віддалений виклик процедур

Технологія *віддаленого виклику процедур* (Remote Procedure Call, RPC) [37, 50, 52, 57] є прикладом синхронного обміну повідомленнями із підтвердженням отримання.

Розглянемо послідовність кроків, необхідних для обміну даними в цьому разі.

1. Операцію send оформляють як виклик процедури із параметрами.
2. Після виклику такої процедури відправник переходить у стан очікування, а дані (ім'я процедури і параметри) доставляються одержувачеві. Одержувач може перебувати на тому самому комп'ютері, чи на віддаленій машині; технологія RPC приховує це. Класичний віддалений виклик процедур передбачає, що процес-одержувач створено внаслідок запиту.
3. Одержувач виконує операцію receive і на підставі даних, що надійшли, виконує відповідні дії (викликає локальну процедуру за іменем, передає їй параметри і обчислює результат).
4. Обчислений результат повертають відправникові як окреме повідомлення.
5. Після отримання цього повідомлення відправник продовжує своє виконання, розглядаючи обчислений результат як наслідок виклику процедури.

Приклади використання віддаленого виклику процедур будуть нами розглянуті в розділі 20.

Висновки

- Потоки різних процесів, що взаємодіють, мають використовувати засоби міжпроцесової взаємодії, завданнями якої є забезпечення обміну даними між захищеними адресними просторами, а також їхня синхронізація. До основних видів міжпроцесової взаємодії належать передавання повідомлень, розподілювана і відображувана пам'ять.
- Головною особливістю передавання повідомлень є те, що ця технологія не вимагає наявності спільно використовуваних даних. Процеси обмінюються повідомленнями змінної довжини за допомогою примітивів send і receive. Ця технологія може бути застосована для організації взаємодії між процесами, виконуваними на віддалених комп'ютерах.

Контрольні запитання та завдання

1. Припустімо, що до комунікаційного каналу, наданого ОС, можуть «підключатися» два процеси. Які синхронізаційні примітиви на рівні ядра ОС можуть бути використані для обміну даними цим каналом відповідно до технології програмних каналів (pipes)? Як приклад ОС візьміть систему Linux.
2. Перелічіть можливі відмінності реалізації черги повідомлень і програмного каналу на рівні ядра ОС.

3. Система обміну повідомленнями надає примітиви send і receive. Примітив receive призупиняє процес, якщо немає повідомлень, призначених для нього. Чи можливе взаємне блокування процесів, якщо не враховувати інших повідомлень, а спільних даних у процесів немає?
4. Реалізуйте систему обміну повідомленнями між потоками на базі стандартних синхронізаційних примітивів розділу 5. Операція `int msg_send(int msg, int priority)` передає в систему повідомлення із заданим пріоритетом. Операція `int msg_recv(int priority)` вилучає із системи найстаріше повідомлення з пріоритетом, більшим або рівним `priority`, і повертає його значення; якщо такого повідомлення немає, поточний потік призупиняють.
5. Перелічіть спільні риси і відмінності поіменованих каналів і сокетів.

Розділ 7

Практичне використання багатопотоковості

- ✦ Проблема взаємних блокувань
- ✦ Запобігання взаємним блокуванням
- ✦ Інверсія пріоритету та інші проблеми багатопотокових застосувань
- ✦ Потоки та організація паралельних обчислень
- ✦ Потоки та моделювання динамічних систем

У цьому розділі ми розглянемо практичні задачі, пов'язані з використанням кількох потоків різного призначення, що взаємодіють. Спочатку докладно зупинимося на виникненні взаємних блокувань у застосуваннях, а потім вивчимо використання потоків для організації паралельних обчислень і моделювання даних.

7.1. Взаємні блокування

У разі використання блокувань може виникнути ситуація *взаємного блокування* (тупик, deadlock), коли жоден із потоків не може продовжувати виконання, бо очікує дій іншого. Вихід із такої ситуації без стороннього втручання неможливий.

Наприклад, взаємне блокування виникає як наслідок таких дій.

1. Потік T_A блокує доступ до ресурсу R_A.
2. Потік T_B блокує доступ до ресурсу R_B.
3. Потік T_A намагається отримати доступ до ресурсу R_B і призупиняється, оскільки той зайнятий потоком T_B.
4. Потік T_B намагається отримати доступ до ресурсу R_A і теж призупиняється, оскільки той зайнятий потоком T_A.

Деякі інші приклади взаємних блокувань вже були наведені (випадок очікування на семафорі всередині критичної секції в розділі 5.3.1 і випадок повторного зайняття м'ютекса потоком у розділі 5.3.2).

У цьому розділі розглянемо особливості виникнення взаємних блокувань, дії для їхнього запобігання і усунення.

7.1.1. Умови виникнення взаємних блокувань

Є чотири умови виникнення взаємних блокувань (коли хоча б одна з них не виконується, взаємні блокування в системі неможливі):

- *взаємне виключення* – ресурсом у конкретний момент може володіти тільки один потік, інші мають очікувати;
- *утримування і очікування* – потік у конкретний момент може володіти кількома ресурсами і робити запити на нові; усі інші потоки мають очікувати, поки він не звільнить доступ до своїх ресурсів;
- *відсутність витиснення* – у потоку не можна відібрати ресурси, якими він володіє, потік має звільнити їх сам;
- *циклічне очікування* – має існувати така циклічна послідовність потоків, за якої кожен попередній очікує доступу до ресурсу, зайнятого наступним (циклічність послідовності полягає в тому, що останній потік чекає ресурсу, зайнятого першим).

Для ілюстрації циклічного очікування захоплення ресурсів часто зображають у вигляді графу (графу розподілу ресурсів), де вузлами є потоки і ресурси (потоки позначають кругами, а ресурси – квадратами), а ребрами – операції захоплення і очікування (ребро, спрямоване від ресурсу до потоку, означає захоплення, від потоку до ресурсу – очікування). Взаємні блокування відповідають циклам у цьому графі (рис. 7.1).

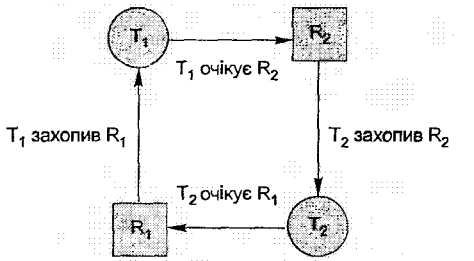

Рис. 7.1. Граф розподілу ресурсів у разі взаємного блокування

7.1.2. Запобігання взаємним блокуванням на рівні ОС

У літературі описані складні алгоритми і стратегії виявлення взаємних блокувань і запобігання ним в ОС, які використовують переважно теорію графів. Більшість таких алгоритмів ґрунтуються на попередніх знаннях про особливості розподілу ресурсів між процесами і тому не мають практичного значення (оскільки в системах, де одночасно виконуються сотні процесів, такої інформації бути не може). Докладніше про це сказано в [44].

У більшості сучасних систем використовують найпростіший можливий алгоритм – «алгоритм страуса»: система не вживає ніяких дій для запобігання взаємним блокуванням, оскільки організація таких дій коштує дорожче, ніж можливі втрати від появи блокувань. Оскільки на практиці блокування з'являються рідко, на них не звертають уваги. Якщо потрібно, відповідне застосування можна явно знищити (докладніше про це буде сказано далі).

7.1.3. Запобігання взаємним блокуванням у багатопотокових застосуваннях

Коли програміст бореться із взаємними блокуваннями у своїй програмі, ситуація зовсім інша — ігнорувати такі блокування не можна, а виявляти їх під час виконання зазвичай вже пізно. Програміст практично завжди здатний тільки запобігати взаємним блокуванням. Розглянемо, які методи можуть бути застосовані для цього.

Обмеження кількості захоплюваних ресурсів

Найпростішою, але не завжди придатною, стратегією є заборона потоку зберігати більше одного ресурсу в кожний момент часу. Тим самим порушується умова утримування і очікування, тому взаємних блокувань у системі бути не може. На жаль, не всі застосування допускають таку заборону, логіка деяких з них вимагає одночасного утримання кількох блокувань.

Частковий порядок блокувань

Складнішим способом запобігання взаємним блокуванням є встановлення часткового порядку блокувань. Для цього потрібно спочатку пронумерувати ресурси, з якими потрібно працювати (R_1, R_2, R_3 ...), а потім у всьому застосуванні захоплювати ресурси у строго зростаючому або спадному порядку (R_1–R_2–R_3 або R_3–R_2–R_1, але не R_1–R_3–R_2).

Частковий порядок називають також *ієрархією блокувань*. Вона є чергою, а не деревом, тому що кожний нащадок у ній повинен мати одного предка, а кожний предок — одного нащадка.

Основою цього методу є те, що під час виконання циклічного очікування у графі захоплення ресурсів мають одночасно бути ребра, спрямовані від ресурсу з меншим номером до потоку, який захопив ресурс із більшим номером, а також від ресурсу з більшим номером до потоку, що захопив ресурс із меншим номером. Такі ребра можна бачити на рис. 7.1.

Коли ресурси будуть захоплені, як описано раніше, цим буде гарантовано, що подібні ребра у графі захоплення ресурсів не з'являться, а отже, циклічного очікування не буде (рис. 7.2). Завжди в конкретний момент часу існуватиме зайнятий ресурс із максимальним номером. Потік, що його захопив, інші ресурси захоплювати не зможе, крім ресурсів з іще більшими номерами, які за визначенням доступні; отже, він зможе завершитися без проблем. Після цього в таку саму ситуацію потрапить інший потік, він теж зможе успішно завершитися і т. д.

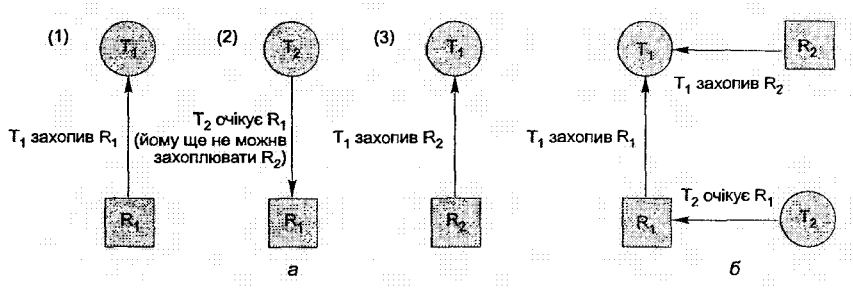

Рис. 7.2. Графи розподілу ресурсів при частковому порядку блокувань: *а* — етапи захоплення ресурсів; *б* — результат захоплення ресурсів (взаємного блокування немає)

Двофазне блокування

Ще одним доволі простим методом запобігання взаємним блокуванням є *двофазне блокування*. При цьому блокування на необхідні ресурси запроваджують по одному; якщо в разі спроби чергового блокування потік виявить, що це вже зроблено іншим потоком, *усі* запроваджені до цього моменту блокування знімають, після чого алгоритм виконують спочатку. Для того щоб під час перевірки самому не перейти у стан очікування, потік має використати неблокувальний виклик на зразок `mutex_trylock()`; запровадивши всі блокування, починають роботу з ресурсами; після закінчення роботи з ресурсами всі блокування знімають.

Недоліком двофазного блокування є висока ціна при великій кількості ресурсів, а також те, що у разі його використання неможливо сховати деталі роботи із ресурсами від коду високого рівня. Якщо виконання дій повторюють, залишається ймовірність того, що й у цьому разі потік не зможе отримати всі ресурси і повторюватиме спроби в циклі, марно витрачаючи процесорний час. Таку ситуацію називають *живим блокуванням* (livelock) [12], воно нічим не краще за взаємне «мертве» блокування (а можливо, навіть гірше, оскільки такі потоки використовують більше ресурсів).

7.1.4. Взаємні блокування і модульність програм

Розглянуті методи запобігання взаємним блокуванням широко застосовують на практиці, однак, на жаль, не вдається розв'язати проблему в цілому, особливо для прикладних програм великого масштабу, які складаються з кількох незалежних модулів. Основна проблема тут полягає в тому, що запобігання взаємним блокуванням порушує модульність програм. Розглянемо це питання докладніше і наведемо ще один приклад виникнення взаємних блокувань — *проблему вкладених моніторів*.

Вона виникає, коли у програмі є два рівні абстракції та функції вищого рівня `high1()` і `high2()` викликають функції нижчого рівня (відповідно `low1()` і `low2()`), при цьому на кожному рівні реалізовано монітор. Наприклад, функція зміни банківського рахунку викликає функцію записування у базу даних, а функція отримання інформації про рахунок — функцію читання з бази даних.

Монітори кожного рівня реалізовані м'ютексами (відповідно M_H і M_L), усередині `low1()` виконують очікування на умовній змінній C_L, усередині `low2()` — її сигналізацію.

```
mutex_t MH, ML;
condition_t CL;
void low1() {
    mutex_lock(ML);
    while(!condition) wait(CL, ML);
    mutex_unlock(ML);
}
void low2() {
    mutex_lock(ML);
    signal(CL, ML);
    mutex_unlock(ML);
}
```

```
void high1() {
    mutex_lock(MH);
    low1();
    mutex_unlock(MH);
}
void high2() {
    mutex_lock(MH);
    low2();
    mutex_unlock(MH);
}
```

Взаємне блокування виникає за такої ситуації.

1. У потоці T_1 викликають `high1()`, блокують м'ютекс M_H; із `high1()` викликають `low1()`, блокують м'ютекс M_L; усередині `low1()` відбувається очікування на C_L, внаслідок чого розблоковується м'ютекс M_L.
2. У потоці T_2 викликають `high2()`, робиться спроба заблокувати м'ютекс M_H; оскільки очікування на умовній змінній C_L не розблокувало M_H, потік переходить у стан очікування, в якому й залишається, поки не продовжить виконання потік T_1; потік T_1 продовжити виконання не може, оскільки нікому сигналізувати C_L (це має робити T_2, але він призупинений і дійти до виклику `low2()` не може).

У результаті обидва потоки заблокували один одного.

Якщо монітори вкладені, взаємне блокування складне для виявлення і налагодження, оскільки код одного рівня може не залежати від коду іншого (наприклад, код `low1()` і `low2()` можуть бути надані розробником бібліотеки). Тобто маємо знати, що робить із блокуваннями код нижнього рівня перед тим, як використати його. Це і є приклад порушення модульності під час запобігання взаємним блокуванням – треба знати тонкощі внутрішньої реалізації низькорівневого коду для того, щоб успішно його використати з коду високого рівня.

Загалом для вирішення цієї проблеми запропоновано такі дії, як зняття блокування на час виклику функцій нижнього рівня і поновлення його після виходу з них, проте, хоча в найпростіших випадках ці дії допомагають, загальна проблема залишається. Припустімо, що одна одну викликають не чотири функції, які можна контролювати, а сотні функцій програмних модулів промислового масштабу, при цьому всі шляхи взаємодії цих модулів перевірити важко (і модулі можуть бути розроблені незалежно). Зрозуміло, у цьому разі забезпечити доступність для одних модулів деталей реалізації інших надзвичайно складно, і запобігання взаємним блокуванням перетворюється в серйозну проблему, загального вирішення якої ще не запропоновано.

7.1.5. Дії у разі виявлення взаємних блокувань

Після виявлення в системі взаємного блокування необхідно вирішити, які дії слід виконувати далі.

1. Можна нічого не робити і дати застосуванню залишатися в такому стані до перезавантаження системи. Це частина «алгоритму страуса», про який уже йшлося і який має право на існування, якщо взаємні блокування з'являються рідко, а боротьба з ними коштує дорожче, ніж втрати від них.

2. Можна знищувати заблоковані потоки і звільняти всі захоплені ними ресурси. Це доволі ризиковано, оскільки неможливо заздалегідь передбачити, у якому стані перебуватимуть такі потоки. Проте на практиці в ОС найчастіше поєднують цей підхід з попереднім, тобто в самій ОС автоматично нічого не виконується, але дають можливість користувачам системи знищувати процеси, зупинені через взаємні виключення, подібно до будь-яких «завислих» процесів.
3. Є й складніші схеми. Наприклад, можна реалізувати *відкат* (rollback), коли застосування зберігає інформацію про свої дії з ресурсами і, якщо з'явиться потреба, на підставі тієї інформації може поновити стан цих ресурсів до взаємного блокування. Місце у програмі, до якого здійснюють відкат, називають *проміжною точкою* (savepoint).

На рівні застосувань подібні схеми реалізувати складно. Можливість реалізації залежить від участі потоків процесу у взаємному блокуванні (якщо задіяні всі потоки, то таке блокування фактично ніде виявляти). Альтернативою є проста стратегія боротьби за живучість програми, яку використовують у системах реального часу. Виділяють окремий потік (його називають *сторожовим потоком* — watchdog thread), який не бере участі в роботі зі спільно використовуваними даними (а, отже, не піддається ризику взаємного блокування), а тільки періодично перевіряє життєздатність інших потоків програми. Якщо сторожовий потік не отримає від них підтвердження виконання упродовж деякого часу, вважають, що вони зупинені, і застосування (чи заблокований потік) або перезапускають, або роблять відкат.

7.2. Інші проблеми багатопотокових застосувань

Під час розробки багатопотокових застосувань необхідно враховувати, крім описаних раніше, інші важливі джерела можливих проблем. У цьому розділі розглянемо три з них: інверсію пріоритету, ступінь деталізації блокувань і відмову в обслуговуванні.

7.2.1. Інверсія пріоритету

Припустимо, що в системі є потоки із різним пріоритетом, які виконуються паралельно і мають спільно використовувати ресурси. Синхронізацію доступу до ресурсів забезпечують блокування. Розглянемо три потоки: T_H — з високим пріоритетом, T_L — із низьким і T_M, пріоритет якого менший, ніж у T_H, але більший, ніж у T_L. Припустимо, що T_H і T_L спільно використовують деякий ресурс. Можлива така ситуація (рис. 7.3).

1. T_L заблоковує ресурс.
2. T_H виходить зі стану очікування, переходить у стан готовності до виконання і намагається отримати доступ до ресурсу. Оскільки ресурс зайнятий потоком T_L, T_H не може отримати керування і переходить у стан очікування.
3. До того, як T_L звільнить ресурс, T_M переходить у стан готовності і як процес із вищим пріоритетом починає виконуватися, витісняючи T_L.

4. Тепер, оскільки T_L не виконується, він не може звільнити ресурс, а отже, процес T_H продовжує залишатися у стані очікування доти, поки процес T_M (із нижчим пріоритетом) продовжуватиме своє виконання.

Таку ситуацію називають *інверсією пріоритету* (priority inversion) [76].

Потік T_H продовжуватиме очікування доти, поки T_L не звільнить ресурс, тому, якщо за час виконання T_M у стан готовності перейдуть нові процеси проміжного пріоритету, вони не дозволять T_L повернути керування, і таке очікування триватиме ще довше.

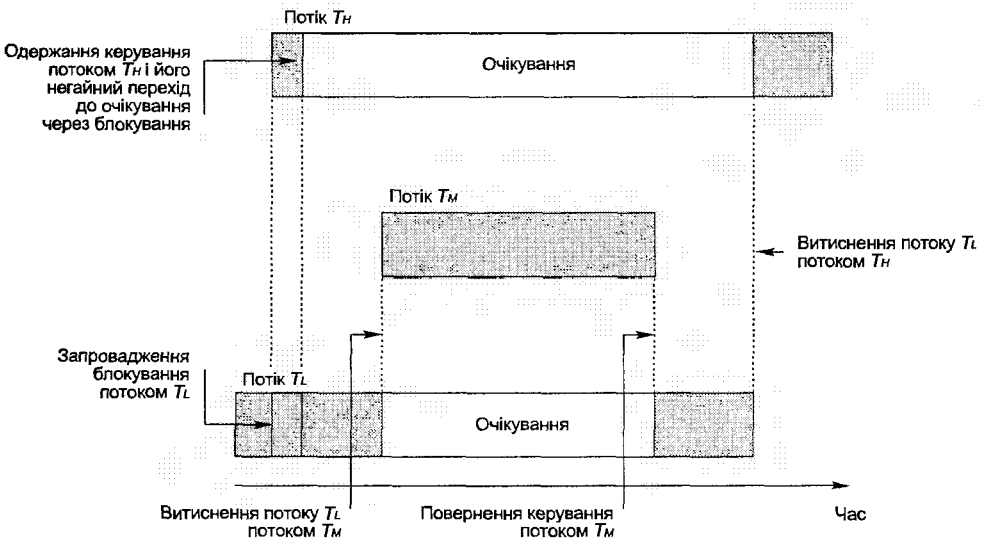

Рис. 7.3. Інверсія пріоритету

Інверсія пріоритету, як і стан змагання, є прикладом помилки, що проявляється тільки в певних умовах, котрі практично неможливо наперед відтворити. Класичним прикладом інверсії пріоритету можна вважати документовану помилку ОС реального часу автоматичного апарата Pathfinder, запущеного для дослідження поверхні Марса в 1997 році. Внаслідок цієї помилки головний потік керування апаратом призупинявся на недопустимий час і ОС апаратно перевантажувалася прямо на Марсі. Проблема полягала у відсиланні головним потоком інформації низькопріоритетній задачі обробки наукових даних через канал. Виходило так, що в той момент, коли ця задача утримувала блокування на канал (призупиняючи тим самим головний потік), її витісняли задачі проміжного пріоритету.

Для розв'язання проблеми інверсії пріоритету можна тимчасово підвищувати пріоритет T_L до пріоритету T_H на той час, поки він утримує блокування на ресурс, потрібний T_H (*спадкування пріоритету*). У цьому разі до розблокування витиснення потоку T_L потоком T_M буде неможливе. Для реалізації цього підходу потрібно пов'язати із кожним ресурсом значення $P_{\max}$ — максимальний пріоритет потоку, якому потрібний цей ресурс, і для кожного потоку, що блокуватиме цей ресурс, підвищувати пріоритет до значення $P_{\max}$ на час, поки він утримує це блокування.

7.2.2. Ступінь деталізації блокувань

Ступінь деталізації блокування (lock granularity) визначає обсяг даних, захищених блокуваннями; що вище цей ступінь, то менше даних захищає кожне блокування. Помилки у визначенні ступеня деталізації блокувань у багатопотокових програмах можуть призвести до зниження продуктивності, появи взаємних блокувань та інших проблем.

Що менше детальними є блокування, то менший ступінь паралелізму припустимий у застосуванні (кількість потоків, які можуть одночасно працювати з даними). Це може негативно впливати на продуктивність застосування. Граничним випадком є стратегія *«одного великого блокування»*, коли весь код застосування або модуля виконують разом з утриманням одного м'ютекса, тобто роботу фактично виконують в однопотоковому режимі.

Якщо в нас, навпаки, ступінь деталізації блокувань високий (є багато дрібних блокувань), ступінь паралелізму теж виявиться високим. При цьому потоки рідше блокують один одного, але застосування стає складнішим, додаються витрати на утримування блокувань і зростає ризик взаємного блокування. Якщо блокування «дроблять», зупинитися потрібно тоді, коли черги потоків, що очікують на м'ютексах, більшу частину часу будуть порожніми і додавання нових блокувань нічого не дасть.

Є різні адаптивні схеми. Наприклад, можна починати із дрібних блокувань, які, коли захопити їх багато, перетворюються на одне велике (*ескалація блокувань*). Можна, навпаки, використати великі блокування, а під час виникнення проблем із паралелізмом розбивати їх на дрібні.

7.2.3. Відмови в обслуговуванні

Відмова в обслуговуванні (denial of service) — ситуація, за якої потік увійде в нескінченний цикл усередині критичної секції, і таким чином назавжди заблокує всі інші потоки на вході в цю критичну секцію, зупинивши, зрештою, застосування у цілому.

Простого засобу розв'язання цієї проблеми немає, як і простої відповіді на низку запитань аналогічного плану, пов'язаних, наприклад, із крахом програми всередині критичної секції тощо. Рекомендовано ретельно розробляти код критичних секцій, щоб таких помилок не допускати. Тут можуть бути корисними неблокувальні виклики типу `mutex_trylock()` і стратегії, подібні до двофазного блокування і сторожового потоку.

7.3. Використання потоків для організації паралельних обчислень

Розглянемо три найвідоміші підходи до організації спільної роботи потоків, виконуваних паралельно [50, 57]. Деякі інші підходи подібного роду (методи «зонд-луна», клітинкових автоматів тощо) можна знайти в [57]. Докладніше особливості організації паралельних обчислень буде розглянуто в розділі 20.

7.3.1. Підхід ведучого-веденого

Найпростішим підходом до організації потоків є підхід ведучого-веденого (master-slave). Основний (ведучий) потік запускає набір ведених потоків і передає кожному з них його обсяг роботи. Він відомий заздалегідь і нарівно розділений між веденими потоками. Після того як ведучий потік запустить всі ведені, він очікує їхнього завершення, виконуючи операцію приєднання потоків. Потім, якщо це необхідно, ведені потоки запускаються знову для виконання роботи наступного етапу.

```
void master() {
    // ініціалізація
    for (i=0; i<n; i++)
        thread_create(slaves[i],slave_fun);
    // (1) своя робота паралельно з веденими
    for (i=0; i<n; i++)
        thread_join(slaves[i]);
    // завершальні дії
}
void slave_fun () {
    // (1) робота
    // результати зазвичай поміщають у глобальну пам'ять
    thread_exit();
}
```

Код, позначений (1), виконується паралельно у ведучому потоці та його ведених.

Цей підхід можна використовувати тоді, коли ведені потоки виконуються незалежно один від одного, тому між ними не потрібна синхронізація. Так можна розв'язувати задачі паралельного виконання ітерацій циклу n потоками за умови, що одна ітерація не залежить від іншої, а їхня кількість m відома заздалегідь. При цьому кожен потік виконує m/n ітерацій. Наведемо приклади використання такого підходу.

1. Перемножування матриць. Потік відповідає за деяку підмножину рядків результуючої матриці, результат обчислень залежить тільки від вихідних матриць, а не від інших потоків.
2. Шифрування даних у режимі електронної кодової книжки (ECB), коли результат отримують шляхом поблочного застосування криптографічного алгоритму до окремих блоків вихідних даних (кожні n байтів вихідного тексту незалежно перетворюються в n байтів шифрованого тексту).
3. Різні статистичні розрахунки (кореляційний, регресійний аналіз тощо).

У разі потреби кількох етапів найпростіший алгоритм ведучого-веденого спричиняється до того, що після закінчення кожного етапу ведені потоки завершуються, а для виконання нового етапу – запускаються заново. Ці операції недостатньо ефективні: хотілося б запустити потоки один раз і синхронізувати їхнє виконання після кожного етапу.

Таку задачу можна розв'язати за допомогою вже знайомої концепції *бар'єра* (див. розділ 5.3.5). Якщо наприкінці кожного етапу ставити бар'єр, спільний для всіх потоків, то замість завершення роботи ведених потоків відбуватимеся їхня

синхронізація на цьому бар'єрі. Після переходу бар'єра ведучий потік може розподілити роботу наступного етапу і повторювати цей процес, поки всю роботу не буде виконано. Наведемо послідовність кроків підходу ведучого-веденого, засновану на використанні бар'єра. Зазначимо, що всі потоки просуваються від етапу до етапу паралельно.

```
void master() {
    barrier_init(barrier, n+1);
    for (i=0; i<n; i++)
        thread_create(slaves[i],slave_fun);
    // поетапна робота
    while (not done) {
        // (1) задання роботи для ведених
        barrier_wait(barrier);
        // (2) своя робота паралельно з веденими
        barrier_wait(barrier);
        // обробка результатів етапу
    }
    for (i=0; i<n; i++)
        thread_join(slaves[i]);
}
void slave_fun () {
    while (not done) {
        // (1) очікування роботи
        barrier_wait(barrier);
        // (2) робота
        // результати зазвичай поміщають в глобальну пам'ять
        barrier_wait(barrier);  // очікування інших
    }
    thread_exit();
}
```

Зазначимо, що ділянки коду, позначені відповідно (1) і (2), для ведучого та веденого виконуються паралельно. У кінці кожної такої ділянки буде операція очікування на спільному для всіх потоків бар'єрі.

7.3.2. Підхід портфеля задач

У разі використання підходу *портфеля задач* (workpile) або *бригади робітників* (work crew) робочі потоки здійснюють запити на свої частини роботи (*одиниці роботи*) із «портфеля задач», який зазвичай є чергою. Під час виконання робочому потокові надають можливість додавати нові одиниці роботи в портфель (для наступного використання іншими потоками). Використання моделі завершується, коли в портфелі не залишається одиниць роботи; керуючий потік може також припинити виконання явно (якщо, наприклад, воно триває занадто довго або, навпаки, результату досягнуто раніше, ніж спорожнів портфель задач).

Основні переваги цього підходу полягають у його простоті, автоматичному розподілі навантаження між робочими потоками, добрій масштабованості. Наприклад, після додавання нових процесорів для ефективнішої роботи достатньо збільшити кількість робочих потоків.

Наведемо деякі галузі застосування портфеля задач.

- Паралельне виконання ітерацій циклу (одиницею роботи є ітерація циклу). Для цієї задачі такий алгоритм ефективніший за метод ведучого-веденого, оскільки забезпечує вищий ступінь паралелізму.
- Алгоритми пошуку найкоротшого шляху (одиницями роботи є окремі ребра графу; ребро з мінімальною довжиною шляху з числа тих, що примикають до цього ребра, додають у портфель), комбінаторні алгоритми на зразок методу гілок і меж, взагалі будь-які алгоритми, що використовують рекурсію (наприклад, обхід каталогу на диску з заходом у підкаталоги).
- Трасування променів (одиницею роботи є піксел або трасувальна лінія), анімація зображень (одиниці роботи — окремі кадри), комп'ютерні ігри (одиниці роботи — становище у грі).

Розглянемо базову послідовність кроків у разі використання портфеля задач.

1. Основний потік створює набір робочих потоків у призупиненому стані, заносить у портфель вихідну роботу і відсилає сигнал про це, поновлюючи один із робочих потоків. Після цього він залишається чекати сигналу про закінчення роботи.
2. Робочий потік забирає одиницю роботи з портфеля (звичайно з початку черги), обробляє її та продовжує чекати нової роботи. Наприклад, під час багатопотокового пошуку у файловій системі роботою може бути ім'я каталогу, а її обробкою — обхід цього каталогу.
3. Під час виконання дій робочому потокові може знадобитися створити нову одиницю роботи. Щоб передоручити частину роботи іншому потокові, він додає її в портфель (звичайно в кінець черги). Прикладом може бути ситуація, коли під час обходу каталогу трапляється підкаталог. Його ім'я можна помістити в портфель, передавши роботу з його обходу іншому потокові.
4. Якщо усі потоки забрали свої роботи з портфеля і жоден не додав нової, останній з потоків, що завершують роботу, відсилає решті сигнал про закінчення роботи.

Загальний вигляд функції робочого потоку для цього випадку буде таким:

```
for( ; ;) {
    while (!workpile_is_empty && !exit) wait();
    if (exit) exit();
    work = get_work(workpile);
    process_work(work);
    if (need_more_work(work))
        add_work(new_work, workpile);
    if (no_more_work) broadcast(exit);
}
```

Наведемо приклад розв'язання задачі з використанням такого підходу. Спочатку зупинимося на структурах даних, які для цього знадобляться.

Спосіб відображення одиниці роботи для портфеля залежить від специфіки задачі. У нашому прикладі така одиниця буде оформлена просто як структура `work_t` із єдиним полем `data`.

Знадобляться також два синхронізаційні примітиви.

1. Загальний м'ютекс для захисту спільно використовуваних даних:

```
mutex_t mutex;
```

2. Умовна змінна changed, сигналізація якої означає, що стан системи змінився, — у портфель додалася нова робота або остання робота в системі виконана, і всі потоки потрібно завершувати. Під час очікування на цій умовній змінній необхідно перевіряти дві умови, оскільки робочий потік *одночасно* очікує, що в портфель прийде робота і що йому надійде сигнал про закінчення:

```
condition_t changed;
```

До спільно використовуваних даних належатимуть лічильник незавершених робіт unfinished_count, який збільшують у разі додавання роботи в портфель і зменшують після закінчення її виконання потоком; прапорець завершення роботи finished, який задають, коли в системі не залишилося незавершених робіт (перед тим, як відіслати сигнал завершення); портфель робіт workpile (його реалізація може бути різною, найчастіше це звичайна черга типу FIFO):

```
volatile int unfinished_count;
volatile int finished;
volatile workpile_t workpile;
```

Найважливішою частиною реалізації підходу є функція робочого потоку. Виокремимо найістотніші особливості її реалізації.

- Усе виконання коду функції відбувається у нескінченному циклі. Насамперед у цьому циклі потік виконує операцію очікування на умовній змінній changed. Це означає, що він очікує появи наступної роботи, якщо портфель порожній; крім того, він може зреагувати на сигнал про вихід, який йому надішле інший потік. Наявність сигналу про вихід перевіряють за увімкнутим прапорцем finished. Якщо такий сигнал є, потік відразу завершують.
- Після виходу з операції очікування і перевірки сигналу завершення в портфелі гарантовано є робота. Потік забирає її з портфеля і обробляє. На час обробки для підвищення паралелізму знімають блокування.
- У разі необхідності потік додає нову роботу в портфель, що звичайно зводиться до додавання елемента у хвіст черги. Після цього сигналізують змінну changed, що призводить до поновлення потоку, який очікує на ній, якщо портфель раніше був порожній.
- Після того як потік завершив необхідні дії, лічильник незавершених робіт unfinished_count зменшують на одиницю. Відмінність перевірки цього лічильника від перевірки кількості робіт у портфелі на початку циклу полягає в тому, що за порожнього портфеля в системі може бути ще багато потоків, які продовжують виконувати свою роботу, тому цей лічильник буде більший від нуля.
- Якщо внаслідок зменшення на одиницю unfinished_count стає рівним нулю, це означає, що справді настав час завершувати роботу, оскільки цей потік останнім виконував корисні дії; у цей момент інші потоки чекають на умовній змінній при вході у цикл. У цьому випадку вмикають прапорець finished, відбувається широкомовна сигналізація умовної змінної (відсилається сигнал закінчення), і виконання функції завершується.

```
void worker_fun () {
    work_t work, new_work;
    for (; ;) {
        mutex_lock (mutex);
        while (is_empty(workpile) && !finished) wait (changed, mutex);
        if (finished) {
            mutex_unlock (mutex); return;
        }
        // взяти нову роботу work із портфеля
        work = remove_from_pile(workpile);
        mutex_unlock (mutex);

        perform_work(work);
        if (new_work_is_needed()) {  // потрібно додати нову роботу
            mutex_lock (mutex);
            // додати нову роботу в портфель
            append_to_pile(workpile, new_work);
            unfinished_count++;
            signal (changed);
            mutex_unlock (mutex);
        }
        // робота з work завершена, наприклад обійдено весь каталог

        mutex_lock (mutex);
        unfinished_count-i;
        if (unfinished_count <= 0) {
            finished = 1;
            broadcast (changed);
            mutex_unlock (mutex);
            return;
        }
        mutex_unlock (mutex);
    }
}
```

Перед використанням цього підходу потрібно створити портфель, надати finished і unfinished_count значення 0, після чого запустити всі робочі потоки. Після завершення використання потрібно виконати операцію приєднання всіх робочих потоків.

На закінчення розглянемо код керування розв'язуванням задачі (функцію workpile_start()). Перед початком виконання роботи є ймовірність того, що не всі потоки завершилися з часу попереднього виклику цієї функції, тому потрібно дочекатися, поки лічильник незакінчених робіт не досягне нуля. Далі слід створити початкову одиницю роботи і додати її в портфель. Після додання вихідного значення почнеться робота (один із потоків поновиться, можливо, додасть свої одиниці роботи в портфель тощо). Після цього потрібно дочекатися закінчення роботи.

```
void workpile_start (data_t init_data) {
    work_t init_work;
    mutex_lock (mutex);
    // чекати закінчення попередніх робіт
    while (unfinished_count > 0) wait (changed, mutex);
    finished = 0;
    // додавання початкової одиниці роботи в портфель (початок роботи)
    append_to_pile(workpile, init_data);
```

```
    unfinished_count++;
    signal (changed);
    // очікувати повного закінчення роботи
    while (!finished) wait (changed, mutex);
    mutex_unlock (mutex);
}
```

У головній програмі між ініціалізацією та очищенням можна запускати процес розв'язування задачі довільну кількість разів.

7.3.3. Підхід конвеєра

У разі використання підходу *конвеєра* (pipeline) задачу передають набору потоків, кожен із яких обробляє дані услід за попереднім потоком і перед тим, як почне обробку наступний, подібно до робітника на конвеєрі. Паралельність алгоритму полягає в тому, що дані можуть переміщатися конвеєром одні за одними, і обробка даних може бути почата задовго до того, як завершиться робота з попередніми. Звичайно способи обробки на кожному етапі конвеєра відрізняються (такий конвеєр називають *гетерогенним*); є також *гомогенні конвеєри*, у яких на кожному етапі виконують той самий код.

Галузі застосування конвеєрної обробки такі.

1. Обробка мультимедіа-інформації (стискання і відновлення відеоінформації, перетворення аудіоінформації). На першому етапі інформацію можна зчитувати, затим обробляти і виконувати.
2. Обробка зображень (зображення проходить конвеєром, етапами якого є різні перетворення).
3. Розподілені обчислення (наприклад, так можна виконувати матричні обчислення, обмінюючись проміжними елементами результуючої матриці).

Конвеєр є узагальненням моделі виробників-споживачів на n потоків, при цьому стандартний зв'язок між виробником і споживачем підтримується між будь-якими двома сусідніми потоками конвеєра.

1. Кожен потік очікує отримання даних, після чого він обробляє їх і намагається передати наступному потокові; якщо той не готовий до цього, потік-відправник чекає.
2. Відіславши дані, потік-відправник поновлює потік-одержувач і продовжує очікування нових даних.

Цикл роботи конвеєра починається з передавання даних його першому потокові, а закінчується тим, що кінцевий одержувач (зазвичай головний потік) чекає отримання даних останнім потоком конвеєра і зчитує ці дані як остаточний результат.

Із кожним етапом конвеєра пов'язують таку інформацію.

- М'ютекс, що захищає його дані.
- Дві умовні змінні: `data_arrived`, сигналізація якої означає, що дані надійшли (тоді можна їх обробляти); `ready`, яку сигналізують, коли потік готовий приймати дані (тоді можна їх відсилати).

- Прапорець наявності даних has_data — умова очікування на умовній змінній data_arrived.
- Безпосередньо самі дані.

Ось приклад структури даних етапу конвеєра:

```
struct stage_t {
    mutex_t         mutex;      // м'ютекс для захисту даних
    condition_t data_arrived;   // дані надійшли
    condition_t ready;          // готовий приймати дані
    int         has_data;       // прапорець наявності даних
    data_t      data;           // дані
};
```

Структура даних усього конвеєра має містити посилання на перший і останній етапи, масив (а краще пов'язаний список) структур етапів, а також прапорець active, який показує, що в конвеєрі є дані.

```
struct pipe_t {
    stage_t     head;           // перший етап
    stage_t     tail;           // останній етап
    stage_t     stages[n];      // масив етапів
    int         active;
};
```

Задачі виконання етапу зводяться до трьох основних дій: очікування отримання даних, обробки даних і відсилання результату далі конвеєром. Відсилання результату виділимо в окрему функцію pipe_send(), вона знадобиться у функції потоку і під час ініціалізації конвеєра, коли буде відіслано дані його першому потоку.

```
// відіслати повідомлення з даними етапу dest
void pipe_send (stage_t dest, data_t data) {
    mutex_lock (dest.mutex);
    // якщо дані в адресата є, дочекатися їхнього споживання
    while (dest.has_data) wait (dest.ready, dest.mutex);
    // ... відіслати data етапу dest
    signal (dest.data_arrived);  // повідомити його про це
    mutex_unlock (dest.mutex);
}
```

У функції потоку необхідно виконати очікування надходження даних від попереднього етапу, їхню модифікацію і відсилання наступному етапові (для чого з цієї функції викликатимемо pipe_send()). Очікування буде завершено, коли інший потік надішле поточному дані. У наведеному коді не показано, як визначити такий потік. Це залежить від структури даних списку етапів; найпростіше зробити цю структуру зв'язним списком.

```
void pipe_stage_fun (stage_t stage) {
    mutex_lock (stage.mutex);
    for(; ;) {  // цикл очікування даних
        // якщо дані не надійшли, дочекатися їх
        while (!stage.has_data) wait (stage.data_arrived, stage.mutex);
        // ... модифікувати дані
        // ... відіслати дані наступному потокові за pipe_send()
```

```
        stage.has_data = 0;   // тепер даних немає
        signal (stage.ready); // і про це сповіщають того, хто чекає
    }
}
```

Для створення конвеєра потрібно сформувати список його структур даних (при цьому бажано, щоб із попереднього потоку було легко визначити наступний) і запустити всі потоки.

```
void pipe_create (pipe_t pipe, int num_stages) {
    // ... створити структури для всіх етапів
    pipe.tail = pipe.stages[num_stages];
    pipe.head = pipe.stages[0];

    // створення всіх потоків
    for (int i = 0; i <= num_stages; i++) {
        thread_create (pipe_stage_fun, pipe.stages[i]);
    }
}
```

Для запуску конвеєра достатньо передати початкове значення його першому потокові.

```
void pipe_start (pipe_t pipe, data_t value) {
    pipe_send (pipe.head, value);
}
```

Після цього почнеться обчислення і передавання даних. Якщо на якихось етапах потоки виконуватимуть роботу швидше, ніж необхідно, вони очікуватимуть, поки інші потоки виконають роботу. Коли якісь потоки почнуть відставати у виконанні роботи, інші їх чекатимуть.

Для збирання результатів потрібно дочекатися отримання даних останнім етапом.

```
data_t pipe_result (pipe_t pipe) {
    stage_t tail = pipe.tail;
    mutex_lock (tail.mutex);
    while (!tail.has_data) wait (tail.data_arrived, tail.mutex);
    int result = tail.data;
    tail.has_data = 0;   // тепер даних на останньому етапі немає
    signal (tail.ready); // і про це сповіщають того, хто чекає
    mutex_unlock (tail.mutex);
    return result;
}
```

Головна програма викликає `pipe_create()`, після цього в циклі — `pipe_start()`, а потім в іншому циклі — `pipe_result()`.

```
void main() {
    pipe_t pipe;
    data_t items[m], res[m];
    int i;
    pipe_create(pipe, n);
    for (i=0; i<m; i++) pipe_start(pipe, items[i]);
    for (i=0; i<m; i++) res[i] = pipe_result(pipe);
}
```

Кількість викликів `pipe_result()` має відповідати кількості викликів `pipe_start()`, інакше простоюватиме або сам конвеєр, або очікування отримання даних з нього.

7.4. Реалізація моделювання динамічних систем

У цьому розділі розглянемо побудову спрощених моделей динамічних систем. Такі моделі описують функціонування складної системи комп'ютерним моделюванням компонентів цієї системи і способів їхньої взаємодії. Використання моделі полягає в запуску програми, що реалізує цю модель (такий запуск називають *прогоном моделі*), і збиранні характеристик виконання.

Типовим прикладом моделей такого типу є комп'ютерні ігри на зразок SimCity або Civilization. Оскільки реальний світ є асинхронним, моделі динамічних систем зручно реалізовувати із застосуванням багатопотоковості, при цьому кожному діючому об'єкту реальної системи відповідає окремий потік.

7.4.1. Приклад моделювання

Моделі, які розглядатимемо, – це сукупність *діючих об'єктів* і *ресурсів*. Діючі об'єкти входять у систему через випадкові проміжки часу (приналежні до заданого діапазону) і намагаються дістати доступ до ресурсів. Ресурси можуть бути надані в обмеженій кількості, тому не всі об'єкти негайно отримують до них доступ. Якщо в цей момент ресурс недоступний, об'єкт переходить у стан очікування. Захопивши ресурс, об'єкт його зберігатиме упродовж випадкового проміжку часу, вибраного із заданого діапазону (для кожного ресурсу такий діапазон задають окремо).

Кожний об'єкт має свій життєвий цикл (він з'являється в системі, виконує деякі дії і залишає систему). Таким об'єктам відповідають потоки, усі їхні дії виконуються у функціях потоку.

Наведемо приклади.

- У разі моделювання роботи універмагу об'єктами будуть покупці, а ресурсами – приміщення універмагу (де може перебувати деяка максимальна кількість покупців) і каса (про яку можна припустити, що вона в конкретний момент часу обслуговує (і пропускає) тільки одного покупця).
- У разі моделювання іспиту об'єктами будуть студенти, а ресурсами – аудиторія і викладач, що приймає іспит.

Видається доцільним створити у функції `main()` один потік для всієї моделі (тобто потік універмагу, іспиту тощо), усередині якого в циклі через випадкові проміжки часу буде створено потоки для об'єктів. Питання генерування випадкових чисел і організації очікування впродовж заданого проміжку часу розглянемо після основного алгоритму.

Кожен потік за час існування повинен отримати доступ до деяких ресурсів. Вони бувають двох типів: доступні в конкретний момент тільки для одного об'єкта (такі, як викладач на іспиті, каса універмагу) і доступні в конкретний момент для деякої кількості об'єктів n (наприклад, вхід в універмаг, аудиторія).

Ресурси першого типу моделюють м'ютекси, а ресурси другого типу – очікування на умовних змінних у разі, коли ресурс зайнятий n об'єктами.

Якщо потік захопив ресурс, він очікує протягом деякого випадкового проміжку часу, потім відпускає ресурс і продовжує роботу.

Цикл створення об'єктів триває доти, поки не закінчиться час моделювання або поки не буде створено всі об'єкти. Після цього нові об'єкти (і відповідні їм потоки) не створюють, а головний потік очікує на умовній змінній, поки всі наявні

об'єкти не завершать свою роботу (тобто поки загальна кількість об'єктів не стане рівною нулю).

Розглянемо можливі параметри моделювання на прикладі моделі універмагу (всі вони цілочислові).

- Універмаг відкритий протягом деякого проміжку часу shop_open_time. Після цього нові покупці перестають з'являтися. Покупці, які перебувають в універмазі до моменту його закриття, мають бути обслужені.
- У кожний момент часу в універмазі можуть перебувати shop_capacity людей, інші мають чекати.
- Покупці з'являються в універмазі через випадковий проміжок часу від 1 до shopper_freq.
- Покупці роблять покупки за випадковий проміжок часу від 1 до max_shopping_time.
- Перед виходом кожен покупець має розрахуватися в касі, що обслуговує в конкретний момент часу одну людину, інші мають чекати.
- Покупець перебуває в касі випадковий проміжок часу від 1 до max_checkout_time. Після розрахунку в касі покупець залишає систему.

Спільно використовуваними даними в системі є змінна total_shoppers, що показує поточну кількість об'єктів у системі.

Для розв'язання цієї задачі використовують умовну змінну entrance_free (сигнал про те, що вхід в універмаг вільний), м'ютекс mutex_checkout (каса) та м'ютекс mutex_shoppers (для захисту спільно використовуваних даних).

Універмаг і кожного покупця відображають потоком. У функції потоку для універмагу розраховують час закінчення моделювання (підсумовуванням поточного часу і shop_open_time); у циклі, що триває до настання цього часу, створюють потоки покупців (що відповідає появі покупців в універмазі); після виходу з циклу розглядуваний потік очікує завершення всіх цих потоків усередині функції wait_for_finish().

```
void wait_for_finish() {
    mutex_lock(mutex_shoppers);
    while (total_shoppers != 0) wait(entrance_free, mutex_shoppers);
    mutex_unlock(mutex_shoppers);
}
```

Потоки покупців створюються як від'єднанні, оскільки для роботи програми неважливий результат їхнього завершення. До того ж заздалегідь невідомо, скільки таких потоків буде в системі, а приєднати їх потрібно всі разом. Синхронізація потоку універмагу із потоками покупців відбуватиметься на умовній змінній.

```
int total_shoppers;
mutex_t mutex_checkout, mutex_shoppers;
condition_t entrance_free;

void shop_fun () {
    thread_t shopper_thread;
    long time_to_close = current_time() + shop_open_time;

    while (current_time() < time_to_close) {
```

```
        thread_create_detached(shopper_thread, shopper_fun);
        sleep(random(1,shopper_freq));   // інтервал створення
    }
    wait_for_finish();
}
```

Тепер перейдемо до потоку покупця. У функції потоку покупець намагається ввійти в універмаг (можливо, очікуючи на умовній змінній). Увійшовши, робить покупки (його потік призупиняють на випадковий час, не більший за max_shopping_time); намагається потрапити до каси (можливо, очікуючи на м'ютексі); призупиняється в касі на час, не більший за max_checkout_time, і відсилає сигнал про вихід із універмагу (можливо, поновлюючи один із потоків, що очікують входу).

```
void shopper_fun () {
    mutex_lock(mutex_shoppers);
    while (total_shoppers == shop_capacity)  // чи можна ввійти?
                    wait(entrance_free, mutex_shoppers);
    total_shoppers++;
    mutex_unlock(mutex_shoppers);
    sleep(random(1,max_shopping_time));      // покупки
    mutex_lock(mutex_checkout);
    sleep(random(1,max_checkout_time));      // очікування в касі
    mutex_unlock (mutex_checkout);

    mutex_lock(mutex_shoppers);
    total_shoppers--;
    signal (entrance_free, mutex_shoppers);  // можна ввійти
    mutex_unlock(mutex_shoppers);
}
```

У головній програмі створюють і приєднують потік універмагу

```
void main () {
    thread_t shop_thread;
    thread_create(shop_thread, shop_fun);
    thread_join(shop_thread);
}
```

Зазначимо, що для організації одночасного доступу до ресурсів кількох діючих об'єктів замість умовних змінних можна використати семафори. Семафор ініціалізують числом, рівним місткості ресурсу (наприклад, максимально допустимою кількістю покупців у магазині). Об'єкт, що звільняє ресурс, збільшує семафор, а той, що займає, зменшує його.

7.4.2. Генерування випадкових чисел і відлік системного часу

У програмах розв'язання задач моделювання необхідно, щоб події відбувалися через деякий випадковий проміжок часу. Для цього потрібно знати, як генерувати випадкові кількості й призупиняти потік на заданий час (наприклад, поки покупець в універмазі робить покупки). Крім того, для визначення тривалості моделювання потрібно вміти відраховувати інтервал часу. Ці питання будуть розглянуті в цьому розділі. Більш загальні питання роботи із системним часом будуть розглянуті в розділі 15.6.

Визначення інтервалу часу у Windows XP

Для визначення інтервалу часу у Win32 API можна використати функцію GetTickCount(), яка повертає кількість мілісекунд, що минули з часу перезавантаження ОС. Точність цього таймера становить приблизно 10 мс.

```
unsigned long mstime = GetTickCount();
```

Для визначення точнішого інтервалу часу застосовують таймер високої роздільної здатності. Для отримання його значення використовують функцію QueryPerformanceCounter():

```
BOOL QueryPerformanceCounter(LARGE_INTEGER *pcount);
```

Тут pcount — поточне значення лічильника в одиницях лічильника (counts). Тип LARGE_INTEGER — це об'єднання структури з 32-бітовими полями LowPart і HighPart та 64-бітового значення QuadPart (доступного в компіляторах з підтримкою 64-бітових цілих).

Для того щоб визначити час у секундах, який минув із деякого моменту, потрібно викликати цю функцію на початку і в кінці інтервалу, обчислити різницю між значеннями і поділити її на величину, повернуту функцією QueryPerformanceFrequency().

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER *pfreq);
```

Функція QueryPerformanceFrequency() повертає частоту лічильника (в одиницях лічильника за секунду) або нуль, якщо такий лічильник не підтримується.

```
LARGE_INTEGER start, end, freq;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&start);
// ... код
QueryPerformanceCounter(&end);
// інтервал у мілісекундах
DWORD timediff = 1000 * (end.LowPart - start.LowPart) / freq.LowPart;
```

Визначення інтервалу часу в Linux

Для визначення інтервалу часу в Linux можна скористатися системним викликом gettimeofday(), основним завданням якого є визначення поточного системного часу:

```
#include<sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Тут tv — покажчик на наперед визначену структуру timeval із полями tv_sec (секунди, що минули з початку «епохи UNIX» — 1.01.1970) і tv_usec (додаткові мікросекунди). Для того щоб отримати загальний час у мікросекундах, можна скористатися таким кодом:

```
struct timeval tv;
gettimeofday(&tv, NULL);
unsigned long time_total = tv.tv_sec*1000000 + tv.tv_usec;
```

Тепер time_total можна порівнювати з іншими значеннями, отриманими аналогічно.

Призупинення потоків на конкретний час

Для призупинення потоку у Win32 використовують функцію Sleep(), що затримує виконання потоку на час, заданий параметром msec (у мілісекундах):

```
void Sleep(long msec);
```

У Linux для цього використовують системний виклик nanosleep(). Він дає змогу задати очікування з точністю до наносекунди (10^{-9} с), але реальна точність інтервалу очікування не так велика:

```
#include<time.h>
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

Тут rqtp — покажчик на структуру, що задає час очікування (з полями tv_sec — секунди, tv_nsec — наносекунди в поточній секунді); rmtp — покажчик на структуру, в якій буде збережено час, що залишився до того, як інтервал очікування буде вичерпаний у разі його переривання за сигналом (при цьому функція поверне –1, і змінна errno прийме значення EINTR):

```
struct timespec ts;
ts.tv_sec = 2;  ts.tv_nsec = 1e+9/2;  // очікування упродовж 2.5 с
nanosleep(&ts, NULL);
```

Генерування випадкових чисел

У стандартній бібліотеці мови С генерування випадкових чисел здійснюють за допомогою функції int rand(). Діапазон чисел — від 0 до RAND_MAX (0x7fff).

Функція rand() у Windows 2000 генерує числа, які не цілком відповідають вимогам випадковості. Для підвищення якості послідовності випадкових чисел використовують функцію void srand(unsigned int seed), що переініціалізує генератор випадкових чисел із використанням початкового значення seed. Для цього можна брати, наприклад, поточний системний час:

```
#include<stdlib.h>
srand (GetTickCount());
random_value = rand();
```

Для генерування якісніших випадкових чисел у системах лінії Windows XP можна скористатися засобами бібліотеки CryptoAPI [56]. У Linux теж доступна функція rand() з тими самими параметрами. Якісніші випадкові числа в цій ОС можна отримати читанням зі спеціального файла пристрою /dev/random. Приклад розв'язання цієї задачі буде наведений у розділі 15. Для генерування випадкових чисел на заданому проміжку від min до max можна використати такий вираз:

```
int r = (random_value % (int)((max + 1) - min)) + min;
```

7.4.3. Особливості задач імітаційного моделювання

У нашому викладі було вжито термін «моделювання динамічних систем», а не «імітаційне моделювання», хоча розглянута задача досить схожа на ті, що можна розв'язати за допомогою традиційних імітаційних моделей [15]. Проте ряд спрощень призводить до істотних відмінностей.

Насамперед усі інтервали очікування в моделі відлічуються за системними годинниками комп'ютера. Це, можливо, дає змогу отримати осмислені результати в системах реального часу (де можна задавати час як параметр функції очікування

і знати, що реальне очікування не відрізнятиметься від цього часу більш ніж на величину допуску), але в ОС загального призначення таке використання системних годинників спричиняється до ненадійних результатів. Річ у тому, що в системі постійно відбуваються події з більшим пріоритетом, ніж у потоків цієї моделі, внаслідок чого ці потоки будуть час від часу непередбачено перериватися, і реальні значення часу очікування відрізнятимуться від заданих.

Справжні імітаційні моделі використовують не реальний, а *модельний час*, коли в програмі організовують внутрішні годинники, і всі компоненти системи періодично синхронізуються за ними. Наприклад, можна підтримувати спеціальний лічильник часу для кожного потоку і, коли він досягне певного значення, синхронізувати всі потоки системи на бар'єрі (а замість `sleep()` збільшувати цей лічильник для потоку на потрібну величину).

Ще одна проблема полягає в тому, що наша модель допускає рівномірний розподіл імовірності настання подій (такий розподіл отримують за допомогою генератора випадкових чисел). На практиці в більшості випадків це не так, і справжні імітаційні моделі використовують складніші закони розподілу (наприклад, частота появи покупців у магазині залежить від дня тижня, часу доби та інших параметрів, час розрахунку в касі — від кількості покупок і кваліфікації касира тощо).

Природно, що в цьому разі неможливо зібрати достатньо осмислені статистичні дані за результатами прогону системи (оскільки такий системний час мало що означає), тоді як імітаційні моделі розглядають засоби збору статистичних результатів як найважливішу складову частину. Проте навіть на таких простих прикладах можна перевірити базову коректність моделі (чи немає в ній взаємних блокувань або голодування, чи отримують об'єкти потрібні їм ресурси тощо).

Висновки

- Важливою практичною проблемою, що виникає під час розробки багатопотокових програм, є взаємні блокування, коли потоки не дають змоги один одному продовжувати виконання. Сучасні ОС звичайно ігнорують такі блокування в прикладних програмах, програмісти ж цього робити не можуть і мають використовувати засоби для їхнього запобігання. Боротьба із взаємними блокуваннями складна і може порушувати модульність програм.
- Під час розробки багатопотокових програм доцільно застосовувати перевірені досвідом підходи, зокрема ведучого-веденого, портфеля задач і конвеєра.
- Багатопотоковість є засобом реалізації моделювання складних систем. При цьому модель подають як сукупність діючих об'єктів і ресурсів; діючим об'єктам відповідають потоки, ресурсам — примітиви синхронізації (м'ютекси, умовні змінні, семафори).

Контрольні запитання та завдання

1. У системі три процеси спільно використовують чотири ресурси, які резервують і звільняють один за одним. Для виконання кожному процесові потрібно не більше двох ресурсів. Процес, який одержав ці два ресурси, завершується. Поясніть, чому в цій ситуації не виникає взаємних блокувань.

2. У системі є 150 одиниць ресурсу, розподілених між трьома процесами так: процес Р1 запитує до 70 одиниць (спочатку виділено 45), процес Р2 – до 60 одиниць (спочатку виділено 40), процес Р3 – до 60 одиниць (спочатку виділено 15). Визначте, чи може виникнути взаємне блокування, якщо запустити процес Р4, що вимагає до 60 одиниць, виділивши йому спочатку:
   а) 25 одиниць;
   б) 35 одиниць?
3. Припустімо, що в комп'ютерній системі, в якій не реалізовано засоби боротьби із взаємними блокуваннями, щомісяця виконуються 5000 задач. Взаємні блокування виникають два рази на місяць, внаслідок кожного такого блокування системний адміністратор повинен вручну знімати 10 задач. Вартість виконання однієї задачі становить 2 одиниці. Було запропоновано встановити у системі схему запобігання взаємним блокуванням. Така схема цілком їх усуне, але при цьому час виконання кожної задачі збільшиться на 10 %. Оцініть доцільність цієї пропозиції, якщо вихідне завантаження процесора складає:
   а) 70 %;
   б) 95 %.
4. Монітор М1 викликає функцію монітора М2. В цій функції відбувається очікування на умовній змінній. Що відбудеться у випадках, якщо:
   а) М2 не викликає функцій жодного іншого монітора; звільняють м'ютекс у М2, але не звільняють у М1;
   б) звільняють м'ютекс як у М2, так і в М1?
5. Задача філософів, що обідають.

   Це класична задача, запропонована Е. Дейкстрою. За круглим столом сидять п'ять філософів, що проводять час у міркуваннях та споживанні їжі. У центрі столу стоїть тарілка рису, на столі лежать п'ять паличок для їжі так, що кожен філософ опиняється між парою паличок (рис. 7.4). Коли філософ розмірковує, він не думає про їжу. Коли ж він зголоднів, то намагається взяти дві палички – ліворуч і праворуч від себе. Якщо хоча б одна паличка зайнята, філософ змушений чекати. Коли філософ взяв обидві палички, він починає їсти. Поївши, він кладе обидві палички на стіл.

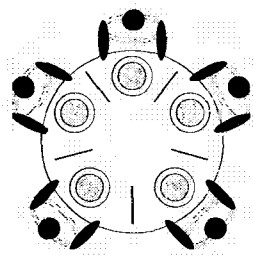

Рис. 7.4. Ілюстрація до задачі філософів, що обідають

Розробіть програму моделювання цієї ситуації. Зверніть увагу на те, що необхідно уникати взаємних блокувань і голодування.

6. Задача сплячого перукаря.

   У залі очікування перукарні з одним робочим місцем є n стільців для клієнтів. Якщо клієнтів немає, перукар відпочиває на одному зі стільців. Якщо клієнт заходить у перукарню і всі стільці в залі очікування зайняті, він іде. Якщо там є вільний стілець, клієнт очікує. Якщо перукар спить, клієнт його будить. Розробіть програму моделювання цієї ситуації.

7. Задача про коктейль.

   За столом сидять господар та троє гостей. Задача гостя — приготувати собі коктейль. Для цього потрібні три складові: сироп, молоко і морозиво. В одного із гостей є сироп, в другого — молоко, у третього — морозиво. У господаря є необмежений запас усіх трьох інгредієнтів. Робочий цикл починається з того, що господар ставить два з трьох компонентів на стіл. Після цього той гість, у якого є відсутній компонент, робить собі коктейль і сповіщає про це господаря. Той у відповідь ставить два інших компоненти на стіл, і цикл повторюється. Розробіть програму моделювання цієї ситуації.

8. Задача про старий міст.

   Міст, рух яким можливий лише в один бік, витримує не більше трьох автомобілів. Автомобілі можуть з'являтися з обох боків мосту. Розробіть алгоритм і програму розв'язання задачі керування рухом по мосту.

9. Розробіть програму множення двох матриць з використанням підходу ведучого-веденого. Кожен ведений потік повинен обчислювати 5 рядків підсумкової матриці.

10. Розробіть програму обходу каталогу імен з використанням підходу портфеля задач. Каталог є списком, кожен елемент якого може бути або ім'ям, або покажчиком на каталог нижнього рівня.

11. Розробіть програму поетапної обробки масиву цілих чисел з використанням підходу конвеєра. Обробка кожного елемента масиву виконується у чотири етапи, на кожному з яких до нього додається число, яке відповідає номеру етапу (від 1 до 4).

12. Користуючись підходом, що розглянуто в розділі 7.4, розробіть програму моделювання роботи аеропорту на проміжку часу T. В аеропорту є n місць для стоянки і єдина злітна смуга, що протягом випадкового проміжку часу (у діапазоні від t_{r1} до t_{r2}) може бути зайнята одним літаком (що йде на зліт або на посадку). Спочатку в аеропорту літаків немає — вони починають прилітати через випадковий проміжок часу (t_{a1}, t_{a2}). Літак, що прилітає, не йде на посадку, поки не звільниться злітна смуга. Після посадки він переміщується на основну стоянку (якщо там є місця) або на тимчасову в очікуванні місця на основній. На основній стоянці літак проводить випадковий час у діапазоні від t_{s1} до t_{s2}, після чого йде на зліт, якщо злітна смуга вільна, або очікує, поки вона звільниться. Після зльоту літак вилучають із системи.

Розділ 8

Керування оперативною пам'яттю

- Означення віртуальної пам'яті
- Принципи адресації пам'яті
- Сегментація пам'яті
- Сторінкова організація пам'яті
- Сторінково-сегментна організація пам'яті
- Керування оперативною пам'яттю в Linux
- Керування оперативною пам'яттю у Windows XP

Під пам'яттю розумітимемо ресурс комп'ютера, призначений для зберігання програмного коду і даних. Пам'ять зображають як масив машинних слів або байтів з їхніми адресами. У фон-неймановській архітектурі комп'ютерних систем процесор вибирає інструкції і дані з пам'яті та може зберігати в ній результати виконання операцій.

Різні види пам'яті організовані в ієрархію. На нижніх рівнях такої ієрархії перебуває дешевша і повільніша пам'ять більшого обсягу, а в міру просування ієрархією нагору пам'ять стає дорожчою і швидшою (а її обсяг стає меншим). Найдешевшим і найповільнішим запам'ятовувальним пристроєм є жорсткий диск комп'ютера. Його називають також допоміжним запам'ятовувальним пристроєм (secondary storage). Швидшою й дорожчою є оперативна пам'ять, що зберігається в мікросхемах пам'яті, встановлених на комп'ютері, — таку пам'ять називатимемо *основною пам'яттю* (main memory). Ще швидшими засобами зберігання даних є різні кеші процесора, а обсяг цих кешів ще обмеженіший.

Керування пам'яттю в ОС — досить складне завдання. Потрібної за характеристиками пам'яті часто виявляють недостатньо, і щоб це не заважало роботі користувача, необхідно реалізовувати засоби координації різних видів пам'яті. Так, сучасні застосування можуть не вміщатися цілком в основній пам'яті, тоді невикористовуваний код застосування може тимчасово зберігатися на жорсткому диску.

У цьому розділі розглянемо технології, які використовують основну пам'ять; керування пам'яттю із застосуванням допоміжних запам'ятовувальних пристроїв буде темою розділу 9, а методи динамічного розподілу пам'яті — розділу 10.

8.1. Основи технології віртуальної пам'яті

Спочатку розглянемо передумови введення концепції віртуальної пам'яті. Наведемо найпростіший з можливих способів спільного використання фізичної пам'яті кількома процесами (рис. 8.1).

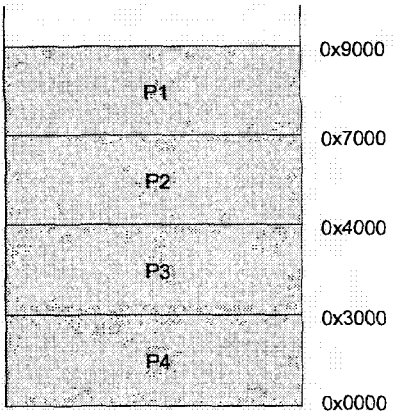

Рис. 8.1. *Спільне використання фізичної пам'яті процесами*

За цієї ситуації кожний процес завантажують у свою власну неперервну ділянку фізичної пам'яті, ділянка наступного процесу починається відразу після ділянки попереднього. На рис. 8.1 праворуч позначені адреси фізичної пам'яті, починаючи з яких завантажуються процеси.

Якщо проаналізувати особливості розподілу пам'яті на основі цього підходу, можуть виникнути такі запитання.

- Як виконувати процеси, котрим потрібно більше фізичної пам'яті, ніж встановлено на комп'ютері?
- Що відбудеться, коли процес виконає операцію записування за невірною адресою (наприклад, процес P2 – за адресою 0x7500)?
- Що робити, коли процесу (наприклад, процесу P1) буде потрібна додаткова пам'ять під час його виконання?
- Коли процес отримає інформацію про конкретну адресу фізичної пам'яті, що з неї розпочнеться його виконання, і як мають бути перетворені адреси пам'яті, використані в його коді?
- Що робити, коли процесу не потрібна вся пам'ять, виділена для нього?

Пряме завантаження процесів у фізичну пам'ять не дає змоги дати відповіді на ці запитання. Очевидно, що потрібні деякі засоби трансляції пам'яті, які давали б змогу процесам використовувати набори адрес, котрі відрізняються від адрес фізичної пам'яті. Перш ніж розібратися в особливостях цих адрес, коротко зупинимося на особливостях компонування і завантаження програм.

Програма зазвичай перебуває на диску у вигляді двійкового виконуваного файла, отриманого після компіляції та компонування. Для свого виконання вона має бути завантажена у пам'ять (адресний простір процесу). Сучасні архітектури

дають змогу процесам розташовуватися у будь-якому місці фізичної пам'яті, при цьому одна й та сама програма може відповідати різним процесам, завантаженим у різні ділянки пам'яті. Заздалегідь невідомо, в яку ділянку пам'яті буде завантажена програма.

Під час виконання процес звертається до різних адрес, зокрема в разі виклику функції використовують її адресу (це адреса коду), а звертання до глобальної змінної відбувається за адресою пам'яті, призначеною для зберігання значення цієї змінної (це адреса даних).

Програміст у своїй програмі звичайно не використовує адреси пам'яті безпосередньо, замість них вживаються символічні імена (функцій, глобальних змінних тощо). Внаслідок компіляції та компонування ці імена прив'язують до переміщуваних адрес (такі адреси задають у відносних одиницях, наприклад «100 байт від початку модуля»). Під час виконання програми переміщувані адреси, своєю чергою, прив'язують до абсолютних адрес у пам'яті. По суті, кожна прив'язка – це відображення одного набору адрес на інший.

До адрес, використовуваних у програмах, ставляться такі вимоги.

- *Захист пам'яті.* Помилки в адресації, що трапляються в коді процесу, повинні впливати тільки на виконання цього процесу. Коли процес P2 зробить операцію записування за адресою 0x7500, то він і має бути перерваний за помилкою. Стратегія захисту пам'яті зводиться до того, що для кожного процесу зберігається діапазон коректних адрес, і кожна операція доступу до пам'яті перевіряється на приналежність адреси цьому діапазону.
- *Відсутність прив'язання до адрес фізичної пам'яті.* Процес має можливості виконуватися незалежно від його місця в пам'яті та від розміру фізичної пам'яті. Адресний простір процесу виділяється як великий статичний набір адрес, при цьому кожна адреса такого набору є переміщуваною. Процесор і апаратне забезпечення повинні мати змогу перетворювати такі адреси у фізичні адреси основної пам'яті (при цьому та сама переміщувана адреса в різний час або для різних процесів може відповідати різним фізичним адресам).

8.1.1. Поняття віртуальної пам'яті

Віртуальна пам'ять – це технологія, в якій вводиться рівень додаткових перетворень між адресами пам'яті, використовуваних процесом, і адресами фізичної пам'яті комп'ютера. Такі перетворення мають забезпечувати захист пам'яті та відсутність прив'язання процесу до адрес фізичної пам'яті.

Завдяки віртуальній пам'яті фізична пам'ять адресного простору процесу може бути фрагментованою, оскільки основний обсяг пам'яті, яку займає процес, більшу частину часу залишається вільним. Є так зване правило «дев'яносто до десяти», або правило локалізації, яке стверджує, що 90 % звертань до пам'яті у процесі припадає на 10 % його адресного простору. Адреси можна переміщати так, щоб основній пам'яті відповідали тільки ті розділи адресного простору процесу, які справді використовуються у конкретний момент.

При цьому невикористовувані розділи адресного простору можна ставити у відповідність повільнішій пам'яті, наприклад простору на жорсткому диску, а в цей час інші процеси можуть використовувати основну пам'ять, у яку раніше відображалися адреси цих розділів. Коли ж розділ знадобиться, його дані завантажують

з диска в основну пам'ять, можливо, замість розділів, які стали непотрібними в конкретний момент (і які, своєю чергою, тепер збережуться на диску). Дані можуть зчитуватися з диска в основну пам'ять під час звертання до них.

У такий спосіб можна значно збільшити розмір адресного простору процесу і забезпечити виконання процесів, що за розміром перевищують основну пам'ять.

8.1.2. Проблеми реалізації віртуальної пам'яті. Фрагментація пам'яті

Основна проблема, що виникає у разі використання віртуальної пам'яті, стосується ефективності її реалізації. Оскільки перетворення адрес необхідно робити під час кожного звертання до пам'яті, недбала реалізація цього перетворення може призвести до найгірших наслідків для продуктивності всієї системи. Якщо для більшості звертань до пам'яті система буде змушена насправді звертатися до диска (який у десятки тисяч разів повільніший, ніж основна пам'ять), працювати із такою системою стане практично неможливо. Питання підвищення ефективності реалізації віртуальної пам'яті буде розглянуто в розділі 9.

Ще однією проблемою є фрагментація пам'яті, що виникає за ситуації, коли неможливо використати вільну пам'ять. Розрізняють *зовнішню і внутрішню фрагментацію пам'яті* (рис. 8.2).

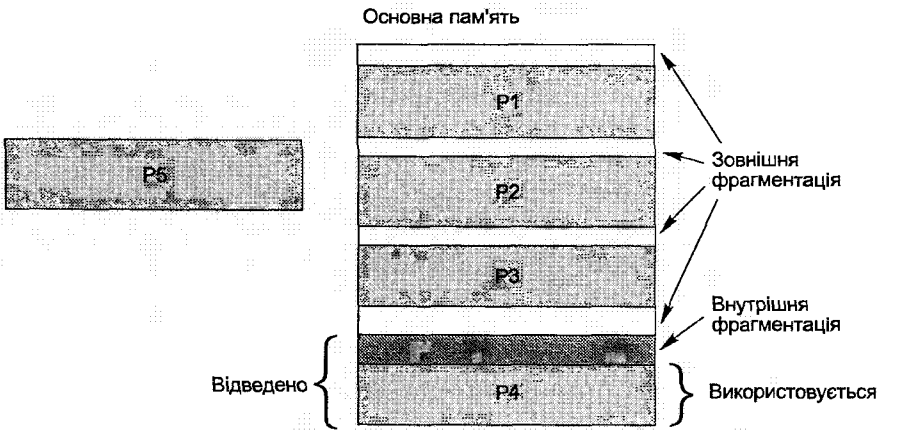

Рис. 8.2. Зовнішня і внутрішня фрагментація

Зовнішня зводиться до того, що внаслідок виділення і наступного звільнення пам'яті в ній утворюються вільні блоки малого розміру – *діри* (holes). Через це може виникнути ситуація, за якої неможливо виділити неперервний блок пам'яті розміру N, оскільки немає жодного неперервного вільного блоку, розмір якого $S \geq N$, хоча загалом обсяг вільного простору пам'яті перевищує N. Так, на рис. 8.2 для виконання процесу P5 місця через зовнішню фрагментацію не вистачає.

Внутрішня фрагментація зводиться до того, що за запитом виділяють блоки пам'яті більшого розміру, ніж насправді будуть використовуватися, у результаті всередині виділених блоків залишаються невикористовувані ділянки, які вже не можуть бути призначені для чогось іншого.

8.1.3. Логічна і фізична адресація пам'яті

Найважливішими поняттями концепції віртуальної пам'яті є логічна і фізична адресація пам'яті.

Логічна або віртуальна адреса – адреса, яку генерує програма, запущена на деякому процесорі. Адреси, що використовують інструкції конкретного процесора, є логічними адресами. Сукупність логічних адрес становить логічний адресний простір.

Фізична адреса – адреса, якою оперує мікросхема пам'яті. Прикладна програма в сучасних комп'ютерах ніколи не має справи з фізичними адресами. Спеціальний апаратний пристрій MMU (memory management unit – пристрій керування пам'яттю) відповідає за перетворення логічних адрес у фізичні. Сукупність усіх доступних фізичних адрес становить фізичний адресний простір. Отже, якщо в комп'ютері є мікросхеми на 128 Мбайт пам'яті, то саме такий обсяг пам'яті адресують фізично. Логічно зазвичай адресують значно більше пам'яті.

Найпростіша схема перетворення адрес зображена на рис. 8.3.

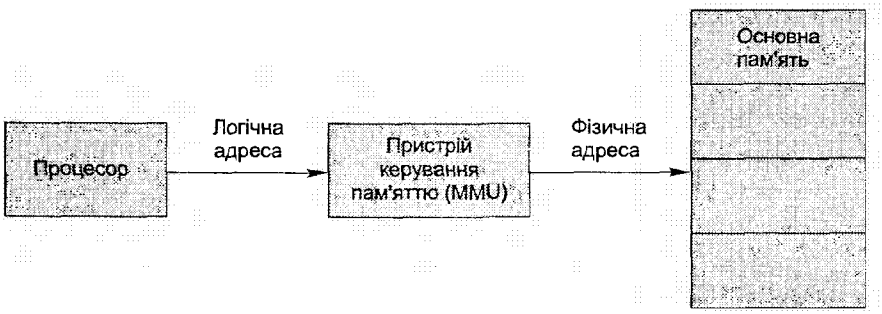

Рис. 8.3. Перетворення логічних адрес пам'яті у фізичні адреси

Специфіку перетворення логічних адрес у фізичні визначають різні підходи до керування оперативною пам'яттю, вивчення яких буде основною темою цього розділу.

8.1.4. Підхід базового і межового регістрів

Під час реалізації віртуальної пам'яті необхідно забезпечити захист пам'яті, переміщення процесів у пам'яті та спільне використання пам'яті кількома процесами.

Одним із найпростіших способів задовольнити ці вимоги є підхід *базового і межового регістрів*. Для кожного процесу в двох регістрах процесора зберігають два значення – *базової адреси* (base) і *межі* (bounds). Кожний доступ до логічної адреси апаратно перетворюється у фізичну адресу шляхом додавання логічної адреси до базової. Якщо отримувана фізична адреса не потрапляє в діапазон (base, base+bounds), вважають, що адреса невірна, і генерують помилку (рис. 8.4).

Такий підхід є найпростішим прикладом реалізації динамічного переміщення процесів у пам'яті. Усі інші підходи, які буде розглянуто в цьому розділі, є різними варіантами розвитку цієї базової схеми. Наприклад, те, що кожний процес у разі використання цього підходу має свої власні значення базового і межового

регістрів, є найпростішою реалізацією концепції адресного простору процесу, яка ґрунтується на тому, що кожний процес має власне відображення пам'яті.

Для організації захисту пам'яті в цій ситуації необхідно, щоб застосування користувача не могли змінювати значення базового і межового регістрів. Достатньо інструкції такої зміни зробити доступними тільки у привілейованому режимі процесора.

До переваг цього підходу належать простота, скромні вимоги до апаратного забезпечення (потрібні тільки два регістри), висока ефективність. Однак сьогодні його практично не використовують через низку недоліків, пов'язаних насамперед з тим, що адресний простір процесу все одно відображається на один неперервний блок фізичної пам'яті: незрозуміло, як динамічно розширювати адресний простір процесу; різні процеси не можуть спільно використовувати пам'ять; немає розподілу коду і даних.

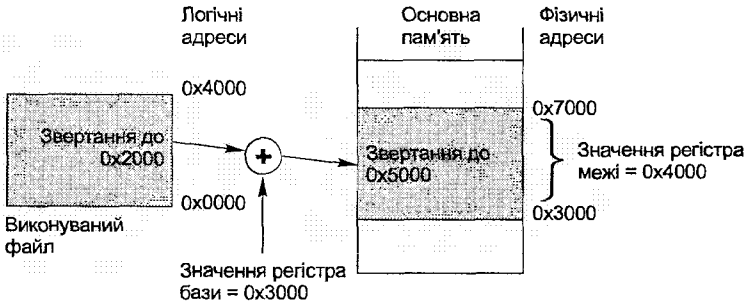

Рис. 8.4. Використання базового і межового регістрів

За такого підходу для процесу виділяють тільки одну пару значень «базова адреса–межа». Природним розвитком цієї ідеї стало відображення адресного простору процесу за допомогою кількох діапазонів фізичної пам'яті, кожен з яких задають власною парою значень базової адреси і межі. Так виникла концепція сегментації пам'яті.

8.2. Сегментація пам'яті

8.2.1. Особливості сегментації пам'яті

Сегментація пам'яті дає змогу зображати логічний адресний простір як сукупність незалежних блоків змінної довжини, які називають *сегментами*. Кожний сегмент звичайно містить дані одного призначення, наприклад в одному може бути стек, в іншому — програмний код і т. д.

У кожного сегмента є ім'я і довжина (для зручності реалізації поряд з іменами використовують номери). Логічна адреса складається з номера сегмента і зсуву всередині сегмента; з такими адресами працює прикладна програма. Компілятори часто створюють окремі сегменти для різних даних програми (сегмент коду, сегмент даних, сегмент стека). Під час завантаження програми у пам'ять створюють таблицю дескрипторів сегментів процесу, кожний елемент якої відповідає одному сегменту і складається із базової адреси, значення межі та прав доступу.

Під час формування адреси її сегментна частина вказує на відповідний елемент таблиці дескрипторів сегментів процесу. Якщо зсув більший, ніж задане значення межі (або якщо права доступу процесу не відповідають правам, заданим для сегмента), то апаратне забезпечення генерує помилку. Коли ж усе гаразд, сума бази і зсуву в разі чистої сегментації дасть у результаті фізичну адресу в основній пам'яті. Якщо сегмент вивантажений на диск, спроба доступу до нього спричиняє його завантаження з диска в основну пам'ять. У підсумку кожному сегменту відповідає неперервний блок пам'яті такої самої довжини, що перебуває в довільному місці фізичної пам'яті або на диску. Загальний підхід до перетворення адреси у разі сегментації показаний на рис. 8.5.

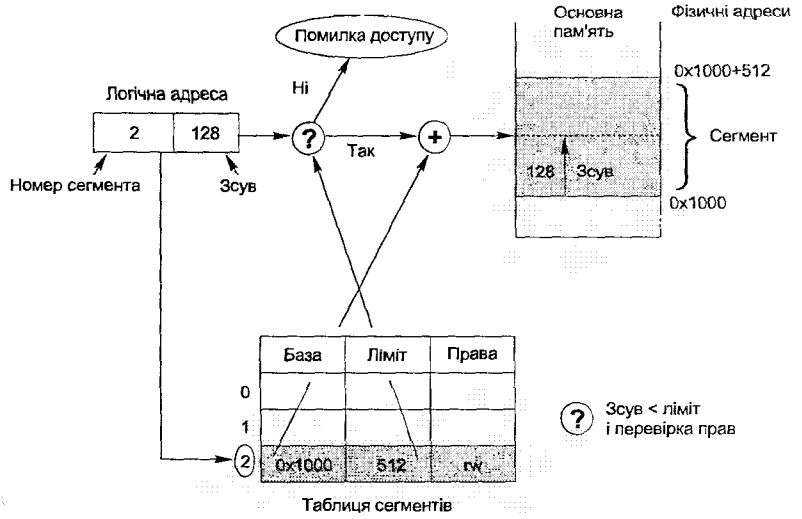

Рис. 8.5. Перетворення адреси у разі сегментації

Загальний вигляд пам'яті у разі сегментації показано на рис. 8.6.

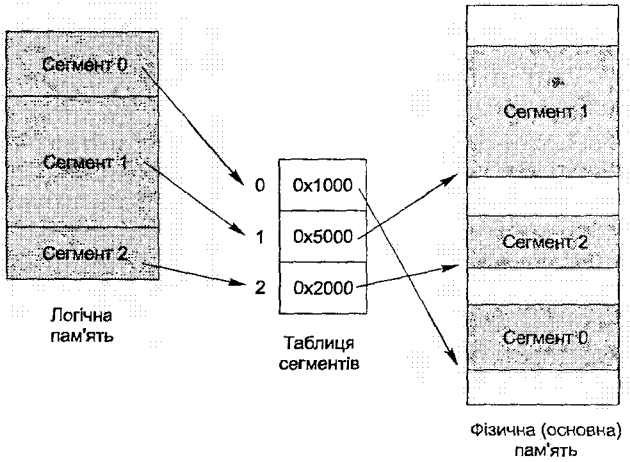

Рис. 8.6. Логічний і фізичний адресний простір у разі сегментації

Наведемо переваги сегментації пам'яті.

- З'явилася можливість організувати кілька незалежних сегментів пам'яті для процесу і використати їх для зберігання даних різної природи. При цьому права доступу до кожного такого сегмента можуть бути задані по-різному.
- Окремі сегменти можуть спільно використовуватися різними процесами, для цього їхні таблиці дескрипторів сегментів повинні містити однакові елементи, що описують такий сегмент.
- Фізична пам'ять, що відповідає адресному простору процесу, тепер не обов'язково має бути неперервною. Справді, сегментація дає змогу окремим частинам адресного простору процесу відображатися не в основну пам'ять, а на диск, і довантажуватися з нього за потребою, забезпечуючи виконання процесів будь-якого розміру.

Цей підхід не позбавлений і недоліків.

- Необхідність введення додаткового рівня перетворення пам'яті спричиняє зниження продуктивності (цей недолік властивий будь-якій повноцінній реалізації віртуальної пам'яті). Для ефективної реалізації сегментації потрібна відповідна апаратна підтримка.
- Керування блоками пам'яті змінної довжини з урахуванням необхідності їхнього збереження на диску може бути досить складним.
- Вимога, щоб кожному сегменту відповідав неперервний блок фізичної пам'яті відповідного розміру, спричиняє зовнішню фрагментацію пам'яті. Внутрішньої фрагментації у цьому разі не виникає, оскільки сегменти мають змінну довжину і завжди можна виділити сегмент довжини, необхідної для виконання програми.

Сьогодні сегментацію застосовують доволі обмежено передусім через фрагментацію і складність реалізації ефективного звільнення пам'яті та обміну із диском. Ширше використання отримав розподіл пам'яті на блоки фіксованої довжини – *сторінкова організація пам'яті*, яку розглянемо в розділі 8.3.

8.2.2. Реалізація сегментації в архітектурі IA-32

В архітектурі IA-32 логічні адреси в програмі формуються із використанням сегментації й мають такий вигляд: «селектор–зсув». Значення селектора завантажують у спеціальний регістр процесора (сегментний регістр) і використовують як індекс у таблиці дескрипторів сегментів, що перебуває в пам'яті та є аналогом таблиці сегментів, описаної раніше. В архітектурі IA-32 підтримуються шість сегментних регістрів. Це означає, що виконуваний код в один і той самий час може адресувати шість незалежних сегментів.

Селектор містить індекс дескриптора в таблиці, біт індикатора локальної або глобальної таблиці та необхідний рівень привілеїв.

Для системи задають спільну *глобальну таблицю дескрипторів* (Global Descriptor Table, GDT), а для кожної задачі – *локальну таблицю дескрипторів* (Local Descriptor Table, LDT).

Дескриптори в IA-32 мають довжину 64 біти. Вони визначають властивості програмних об'єктів (наприклад, сегментів пам'яті або таблиць дескрипторів).

Дескриптор містить значення бази (base), яке відповідає адресі об'єкта (наприклад, початок сегмента); значення межі (limit); тип об'єкта (сегмент, таблиця дескрипторів тощо); характеристики захисту.

Звертання до таблиць дескрипторів підтримується апаратно. Якщо задані в дескрипторі характеристики захисту не відповідають рівню привілеїв, визначеному селектором, отримати доступ до пам'яті за його допомогою буде неможливо. Так забезпечують захист пам'яті.

Проте жодного разу не було згадано, що в дескрипторі зберігають фізичну адресу. Річ у тому, що для архітектури IA-32 внаслідок перетворення логічної адреси отримують не фізичну адресу, а ще один вид адреси, який називають лінійною адресою. У розділі 8.4 розглянемо таке дворівневе перетворення адреси.

8.3. Сторінкова організація пам'яті

До основних технологій реалізації віртуальної пам'яті крім сегментації належить *сторінкова організація пам'яті* (paging). Її головна ідея – розподіл пам'яті блоками фіксованої довжини. Такі блоки називають *сторінками*.

Ця технологія є найпоширенішим підходом до реалізації віртуальної пам'яті в сучасних операційних системах.

8.3.1. Базові принципи сторінкової організації пам'яті

У разі сторінкової організації пам'яті логічну адресу називають також *лінійною*, або *віртуальною*, адресою. Такі адреси належать одній множині (наприклад, лінійною адресою може бути невід'ємне ціле число довжиною 32 біти).

Фізичну пам'ять розбивають на блоки фіксованої довжини – *фрейми*, або сторінкові блоки (frames). Логічну пам'ять, у свою чергу, розбивають на блоки такої самої довжини – *сторінки* (pages). Коли процес починає виконуватися, його сторінки завантажуються в доступні фрейми фізичної пам'яті з диска або іншого носія.

Сторінкова організація пам'яті повинна мати апаратну підтримку. Кожна адреса, яку генерує процесор, ділиться на дві частини: *номер сторінки і зсув сторінки*. Номер сторінки використовують як індекс у таблиці сторінок.

Таблиця сторінок – це структура даних, що містить набір елементів (page-table entries, PTE), кожен із яких містить інформацію про номер сторінки, номер відповідного їй фрейму фізичної пам'яті (або беспосередньо його базову адресу) та права доступу. Номер сторінки використовують для пошуку елемента в таблиці. Після його знаходження до базової адреси відповідного фрейму додають зсув сторінки, чим і визначають фізичну адресу (рис. 8.7).

Розмір сторінки є ступенем числа 2, у сучасних ОС використовують сторінки розміром від 2 до 8 Кбайт. У спеціальних режимах адресації можна працювати зі сторінками більшого розміру.

Для кожного процесу створюють його власну таблицю сторінок. Коли процес починає своє виконання, ОС розраховує його розмір у сторінках і кількість фреймів у фізичній пам'яті. Кожну сторінку завантажують у відповідний фрейм, після чого його номер записують у таблицю сторінок процесу.

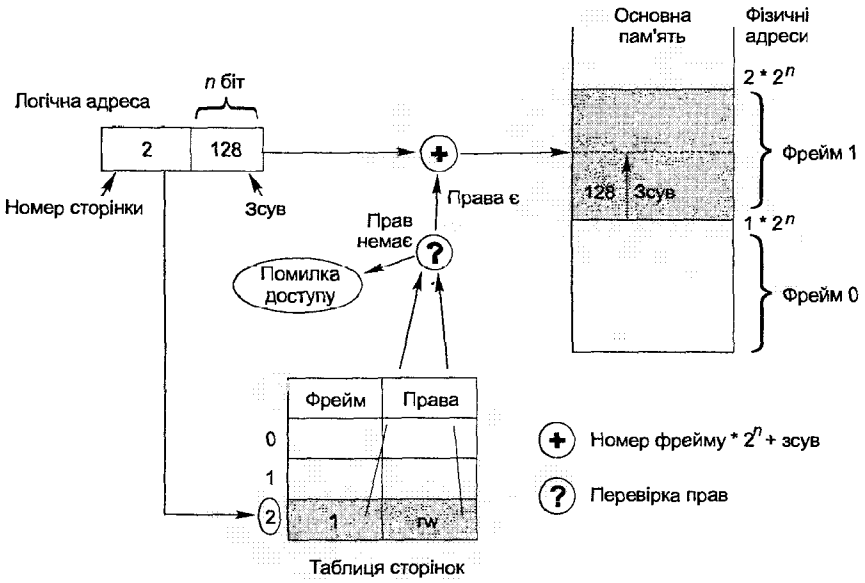

Рис. 8.7. Перетворення адреси у разі сторінкової організації пам'яті

Відображення логічної пам'яті для процесу відрізняється від реального стану фізичної пам'яті. На логічному рівні для процесу вся пам'ять зображується неперервним блоком і належить тільки цьому процесові, а фізично вона розосереджена по адресному простору мікросхеми пам'яті, чергуючись із пам'яттю інших процесів (рис. 8.8). Процес не може звернутися до пам'яті, адреса якої не вказана в його таблиці сторінок (так реалізований захист пам'яті).

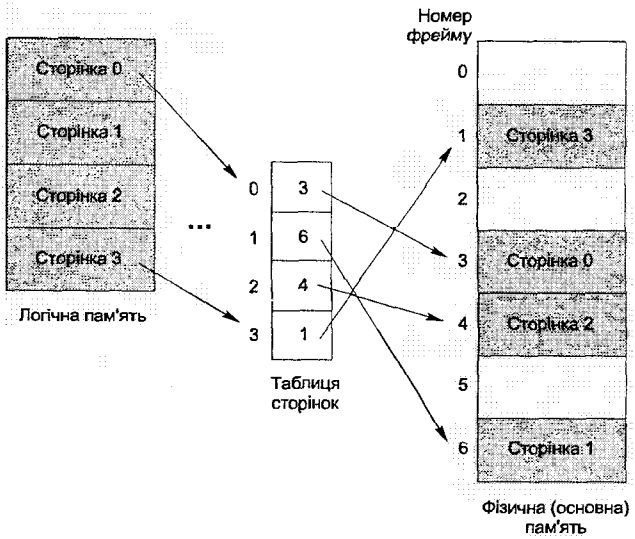

Рис. 8.8. Логічний і фізичний адресний простір у разі сторінкової організації пам'яті

8.3.3. Багаторівневі таблиці сторінок

Щоб адресувати логічний адресний простір значного обсягу за допомогою однієї таблиці сторінок, її доводиться робити дуже великою. Наприклад, в архітектурі IA-32 за стандартного розміру сторінки 4 Кбайт (для адресації всередині такої сторінки потрібні 12 біт) на індекс у таблиці залишається 20 біт, що відповідає таблиці сторінок на 1 мільйон елементів.

Щоб уникнути таких великих таблиць, запропоновано технологію *багаторівневих таблиць сторінок*. Таблиці сторінок самі розбиваються на сторінки, інформацію про які зберігають в таблиці сторінок верхнього рівня. Кількість рівнів рідко перевищує 2, але може доходити й до 4.

Коли є два рівні таблиць, логічну адресу розбивають на індекс у таблиці верхнього рівня, індекс у таблиці нижнього рівня і зсув.

Ця технологія має дві основні переваги. По-перше, таблиці сторінок стають менші за розміром, тому пошук у них можна робити швидше. По-друге, не всі таблиці сторінок мають перебувати в пам'яті у конкретний момент часу. Наприклад, якщо процес не використовує якийсь блок пам'яті, то вміст усіх сторінок нижнього рівня невикористовуваного блоку може бути тимчасово збережений на диску.

8.3.4. Реалізація таблиць сторінок в архітектурі IA-32

Архітектура IA-32 використовує дворівневу сторінкову організацію, починаючи з моделі Intel 80386.

Таблицю верхнього рівня називають *каталогом сторінок* (page directory), для кожної задачі повинен бути заданий окремий каталог сторінок, фізичну адресу якого зберігають у спеціальному керуючому регістрі cr3 і куди він автоматично завантажується апаратним забезпеченням при перемиканні контексту. Таблицю нижнього рівня називають просто *таблицею сторінок* (page table).

Лінійна адреса поділяється на три поля:

- *каталогу* (Directory) – визначає елемент каталогу сторінок, що вказує на потрібну таблицю сторінок;
- *таблиці* (Table) – визначає елемент таблиці сторінок, що вказує на потрібний фрейм пам'яті;
- *зсуву* (Offset) – визначає зсув у межах фрейму, що у поєднанні з адресою фрейму формує фізичну адресу.

Розмір полів каталогу і таблиці становить 10 біт, що дає таблиці сторінок, які містять 1024 елементи, розмір поля зсуву – 12 біт, що дає сторінки і фрейми розміром 4 Кбайт. Одна таблиця сторінок нижнього рівня адресує 4 Мбайт пам'яті (1 Мбайт фреймів), а весь каталог сторінок – 4 Гбайт.

Елементи таблиць сторінок всіх рівнів мають однакову структуру. Виокремимо такі поля елемента:

- *прапорець присутності* (Present), дорівнює одиниці, якщо сторінка перебуває у фізичній пам'яті (їй відповідає фрейм); рівність цього прапорця нулю означає, що сторінки у фізичній пам'яті немає, при цьому операційна система може використати інші поля елемента для своїх цілей;

- *20 найбільш значущих бітів*, які задають початкову адресу фрейму, кратну 4 Кбайт (може бути задано 1 Мбайт різних початкових адрес);
- *прапорець доступу* (Accessed), який покладають рівним одиниці під час кожного звертання пристрою сторінкової підтримки до відповідного фрейму;
- *прапорець зміни* (Dirty), який покладають рівним одиниці під час кожної операції записування у відповідний фрейм;
- *прапорець читання-записування* (Read/Write), що задає права доступу до цієї сторінки або таблиці сторінок (для читання і для записування або тільки для читання);
- *прапорець привілейованого режиму* (User/Supervisor), який визначає режим процесора, необхідний для доступу до сторінки. Якщо цей прапорець дорівнює нулю, сторінка може бути адресована тільки із привілейованого режиму, якщо одиниці – доступна також і з режиму користувача;

Прапорці присутності, доступу і зміни можна використовувати ОС для організації віртуальної пам'яті. Про використання прапорців присутності та зміни говоритимемо у розділі 9.3.1, а про використання прапорця доступу – у розділі 9.5.5.

8.3.5. Асоціативна пам'ять

Під час реалізації таблиць сторінок для отримання доступу до байта фізичної пам'яті доводиться звертатися до пам'яті кілька разів. У разі використання дворівневих сторінок потрібні *три* операції доступу: до каталогу сторінок, до таблиці сторінок і безпосередньо за адресою цього байта, а для трирівневих таблиць – *чотири* операції. Це сповільнює доступ до пам'яті та знижує загальну продуктивність системи.

Як уже зазначалося, правило «дев'яносто до десяти» свідчить, що більша частина звертань до пам'яті процесу належить до малої підмножини його сторінок, причому склад цієї підмножини змінюється досить повільно. Засобом підвищення продуктивності у разі сторінкової організації пам'яті є кешування адрес фреймів пам'яті, що відповідають цій підмножині сторінок.

Для розв'язання цієї проблеми було запропоновано технологію *асоціативної пам'яті* або *кеша трансляції* (translation look-aside buffers, TLB). У швидкодіючій пам'яті (швидшій, ніж основна пам'ять) створюють набір із кількох елементів (різні архітектури відводять під асоціативну пам'ять від 8 до 2048 елементів, в архітектурі IA-32 таких елементів до Pentium 4 було 32, починаючи з Pentium 4 – 128). Кожний елемент кеша трансляції відповідає одному елементу таблиці сторінок.

Тепер під час генерування фізичної адреси спочатку відбувається пошук відповідного елемента таблиці в кеші (в IA-32 – за полем каталогу, полем таблиці та зсуву), і якщо він знайдений, стає доступною адреса відповідного фрейму, що негайно можна використати для звертання до пам'яті. Якщо ж у кеші відповідного елемента немає, то доступ до пам'яті здійснюють через таблицю сторінок, а після цього елемент таблиці сторінок зберігають в кеші замість найстарішого елемента (рис. 8.9).

На жаль, у разі перемикання контексту в архітектурі IA-32 необхідно очистити весь кеш, оскільки в кожного процесу є своя таблиця сторінок, і ті ж самі номери сторінок для різних процесів можуть відповідати різним фреймам у фізичній

пам'яті. Очищення кеша трансляції є дуже повільною операцією, якої треба всіляко уникати (у розділі 8.5.5 буде показано, як цю проблему вирішують у Linux).

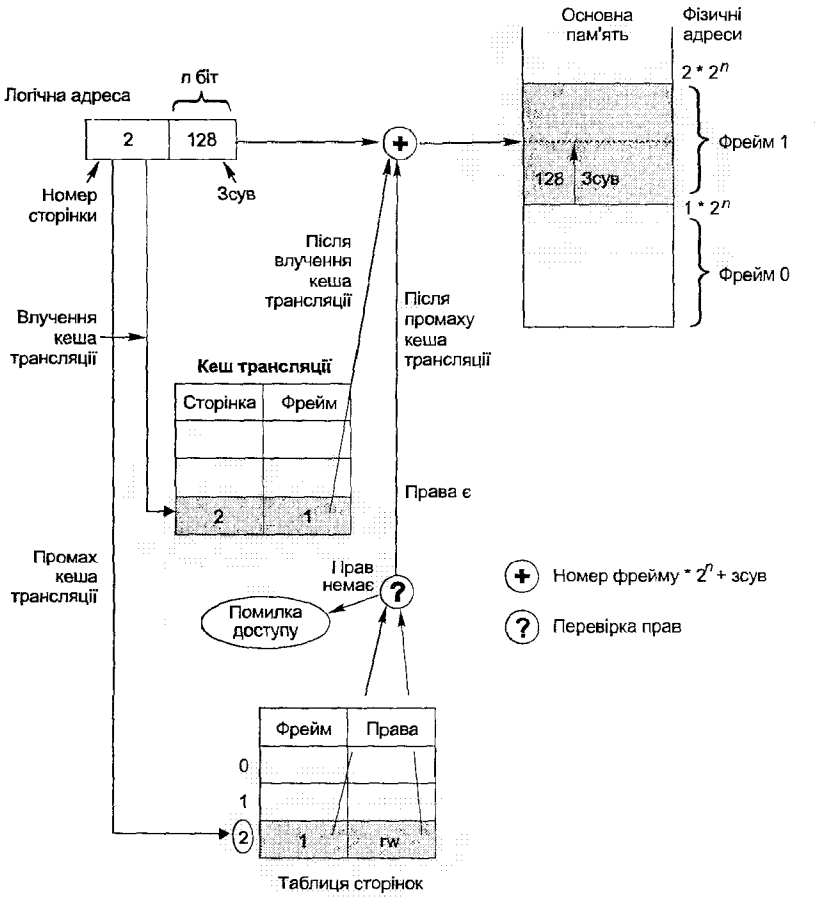

Рис. 8.9. Принцип роботи кеша трансляції

Важливою характеристикою кеша трансляції є відсоток влучень, тобто відсоток випадків, коли необхідний елемент таблиці сторінок перебуває в кеші і не потребує доступу до пам'яті. Відомо, що при 32 елементах забезпечується 98 % влучень. Зазначимо також, що за такого відсотку влучень зниження продуктивності у разі використання дворівневих таблиць сторінок порівняно з однорівневими становить 28 %, однак переваги, отримувані під час розподілу пам'яті, роблять таке зниження допустимим.

8.4. Сторінково-сегментна організація пам'яті

Базові принципи

Оскільки сегменти мають змінну довжину і керувати ними складніше, чиста сегментація зазвичай не настільки ефективна, як сторінкова організація. З іншого

боку, видається цінною сама можливість використати сегменти як блоки пам'яті різного призначення змінної довжини.

Для того щоб об'єднати переваги обох підходів, у деяких апаратних архітектурах (зокрема, в IA-32) використовують комбінацію сегментної та сторінкової організації пам'яті. За такої організації перетворення логічної адреси у фізичну відбувається за три етапи.

1. У програмі задають логічну адресу із використанням сегмента і зсуву.
2. Логічну адресу перетворюють у лінійну (віртуальну) адресу за правилами, заданими для сегментації.
3. Віртуальну адресу перетворюють у фізичну за правилами, заданими для сторінкової організації.

Таку архітектуру називають *сторінково-сегментною організацією пам'яті.*

Перетворення адрес в архітектурі IA-32

Розглянемо особливості реалізації описаних трьох етапів перетворення адреси в архітектурі IA-32.

1. Машинна мова архітектури IA-32 (а, отже, будь-яка програма, розроблена для цієї архітектури) оперує логічними адресами. Логічна адреса, як було зазначено раніше, складається із селектора і зсуву.
2. Лінійна або віртуальна адреса – це ціле число без знака завдовжки 32 біти. За його допомогою можна дістати доступ до 4 Гбайт комірок пам'яті. Перетворення логічної адреси в лінійну відбувається всередині *пристрою сегментації* (segmentation unit) за правилами перетворення адреси на базі сегментації, описаними раніше.
3. Фізичну адресу використовують для адресації комірок пам'яті в мікросхемах пам'яті. Її теж зображають 32-бітовим цілим числом без знака. Перетворення лінійної адреси у фізичну відбувається всередині *пристрою сторінкової підтримки* (paging unit) за правилами для сторінкової організації пам'яті (лінійну адресу розділяють апаратурою на адресу сторінки і сторінковий зсув, а потім перетворюють у фізичну адресу із використанням таблиць сторінок, кеша трансляції тощо).

Формування адреси у разі сторінково-сегментної організації пам'яті показане на рис. 8.10.

Необхідність підтримки сегментації в IA-32 значною мірою є даниною традиції (це пов'язано з необхідністю зворотної сумісності зі старими моделями процесорів, у яких була відсутня підтримка сторінкової організації пам'яті). Сучасні ОС часто обходять таку сегментну організацію майже повністю, використовуючи в системі лише кілька загальних сегментів, причому кожен із них задають селектором, у дескрипторі якого поле base дорівнює нулю, а поле limit – максимальній адресі лінійної пам'яті. Зсув логічної адреси завжди буде рівний лінійній адресі, а отже, лінійну адресу можна буде формувати у програмі, фактично переходячи до чисто сторінкової організації пам'яті. Опишемо такі підходи, коли йтиметься про керування пам'яттю в Linux і Windows XP, у розділах 8.5 та 8.6.

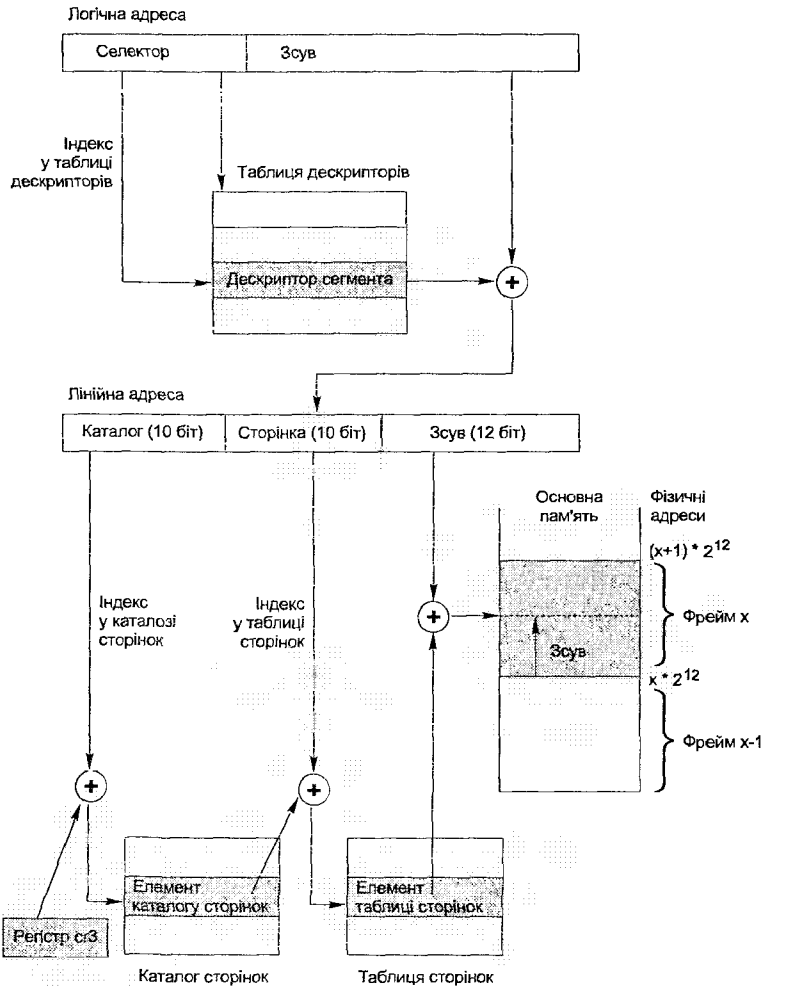

Рис. 8.10. Сторінково-сегментна організація пам'яті в архітектурі IA-32

8.5. Реалізація керування основною пам'яттю: Linux

У цьому розділі розглянемо особливості керування основною пам'яттю Linux версії ядра 2.4.

8.5.1. Використання сегментації в Linux. Формування логічних адрес

Як уже зазначалося, необхідність підтримки сегментації призводить до того, що програми стають складнішими, оскільки задача виділення сегментів і формування коректних логічних адрес лягає на програміста. Цю проблему в Linux вирішують

доволі просто — ядро практично не використовує засобів підтримки сегментації архітектури IA-32. У системі підтримують мінімальну кількість сегментів, без яких неможлива коректна адресація пам'яті процесором (сегменти коду і даних ядра та режиму користувача). Код ядра і режиму користувача спільно використовує ці сегменти.

Сегменти коду використовують під час формування логічних адрес коду (для виклику процедур тощо); такі сегменти позначають як доступні для читання і виконання. Сегменти даних призначені для формування логічних адрес даних (глобальних змінних, стека тощо) і позначаються як доступні для читання і записування. Сегменти режиму користувача доступні з режиму користувача, сегменти ядра — тільки з режиму ядра.

Усі сегменти, які використовуються у Linux, визначають межу зсуву, що дає змогу створити в рамках кожного з них 4 Гбайт логічних адрес. Це означає, що Linux фактично передає всю роботу з керування пам'яттю на рівень перетворення між лінійними і фізичними адресами (оскільки кожна логічна адреса відповідає лінійній).

Далі в цьому розділі вважатимемо логічні адреси вже сформованими (на базі відповідного сегмента) і перетвореними на лінійні адреси.

8.5.2. Сторінкова адресація в Linux

У ядрі Linux версії 2.4 використовують трирівневу організацію таблиць сторінок. Підтримуються три типи таблиць сторінок: *глобальний* (Page Global Directory, PGD); *проміжний каталог сторінок* (Page Middle Directory, PMD); *таблиця сторінок* (Page Table).

Кожний глобальний каталог містить адреси одного або кількох проміжних каталогів сторінок, а ті, своєю чергою, — адреси таблиць сторінок. Елементи таблиць сторінок (PTE) вказують на фрейми фізичної пам'яті.

Кожний процес має свій глобальний каталог сторінок і набір таблиць сторінок. Під час перемикання контексту Linux зберігає значення регістра cr3 у керуючому блоці процесу, що передає керування, і завантажує в цей регістр значення з керуючого блоку процесу, що починає виконуватися. Отже, коли процес починає виконуватися, пристрій сторінкової підтримки вже посилається на коректний набір таблиць сторінок.

Тепер розглянемо роботу цієї трирівневої організації для архітектури IA-32, яка дає можливість мати тільки два рівні таблиць. Насправді ситуація досить проста — проміжний каталог таблиць оголошують порожнім, водночас його місце в ланцюжку покажчиків зберігають для того, щоб той самий код міг працювати для різних архітектур. У цьому разі PGD відповідає каталогу сторінок IA-32 (його елементи містять адреси таблиць сторінок), а під час роботи із покажчиком на PMD насправді працюють із покажчиком на відповідний йому елемент PGD, відразу отримуючи доступ до таблиці сторінок. Між таблицями сторінок Linux і таблицями сторінок IA-32 завжди дотримується однозначна відповідність.

Для платформо-незалежного визначення розміру сторінки в Linux використовують системний виклик getpagesize():

```
#include <unistd.h>
printf("Розмір сторінки: %d\n", getpagesize());
```

8.5.3. Розташування ядра у фізичній пам'яті

Ядро Linux завантажують у набір зарезервованих фреймів пам'яті, які заборонено вивантажувати на диск або передавати процесу користувача, що захищає код і дані ядра від випадкового або навмисного ушкодження.

Завантаження ядра починається із другого мегабайта пам'яті (перший мегабайт пропускають, тому що в ньому є ділянки, які не можуть бути використані, наприклад відеопам'ять текстового режиму, код BIOS тощо). Із ядра завжди можна визначити фізичні адреси початку та кінця його коду і даних.

На рис. 8.11 видно, як розташовується ядро у фізичній пам'яті, а також межі різних ділянок пам'яті ядра.

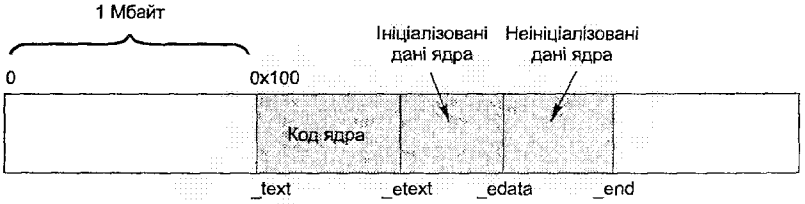

Рис. 8.11. Розташування ядра Linux у фізичній пам'яті

8.5.4. Особливості адресації процесів і ядра

Лінійний адресний простір кожного процесу поділяють на дві частини: перші 3 Гбайт адрес використовують у режимі ядра та користувача; вони відображають захищений адресний простір процесу; решту 1 Гбайт адрес використовують тільки в режимі ядра.

Елементи глобального каталогу процесу, що визначають адреси до 3 Гбайт, можуть бути задані самим процесом, інші елементи мають бути однаковими для всіх процесів і задаватися ядром.

Потоки ядра (див. розділ 3) не використовують елементів глобального каталогу першого діапазону. На практиці, коли відбувається передавання керування потоку ядра, не змінюється значення регістра cr3, тобто потік ядра використовує таблиці сторінок процесу користувача, що виконувався останнім (оскільки йому потрібні тільки елементи, доступні в режимі ядра, а вони в усіх процесах користувача однакові).

Адресний простір ядра починається із четвертого гігабайта лінійної пам'яті. Для прямого відображення на фізичні адреси доступні перші 896 Мбайт цього простору (128 Мбайт, що залишилися, використовується переважно для динамічного розподілу пам'яті ядром).

8.5.5. Використання асоціативної пам'яті

Під час роботи з асоціативною пам'яттю основне завдання ядра полягає у зменшенні потреби її очищення. Для цього вживають таких заходів.

- Під час планування невелику перевагу має процес, який використовує той самий набір таблиць сторінок, що й процес, який повертає керування (під час перемикання між такими процесами очищення кеша трансляції не відбувається).

- Реалізація потоків ядра, котрі використовують таблиці сторінок останнього процесу, теж призводить до того, що під час перемикання між процесом і потоком ядра очищення не відбувається.

8.6. Реалізація керування основною пам'яттю: Windows XP

8.6.1. Сегментація у Windows XP

Система Windows XP використовує загальні сегменти пам'яті подібно до того, як це робиться в Linux. Для всіх сегментів у програмі задають однакові значення бази і межі, тому роботу з керування пам'яттю аналогічним чином передають на рівень лінійних адрес (які є зсувом у цих загальних сегментах).

8.6.2. Сторінкова адресація у Windows XP

Під час роботи з лінійними адресами у Windows XP використовують дворівневі таблиці сторінок, повністю відповідні архітектурі IA-32 (див. розділ 8.3.4). У кожного процесу є свій каталог сторінок, кожен елемент якого вказує на таблицю сторінок. Таблиці сторінок усіх рівнів містять по 1024 елементи таблиці сторінок, кожний такий елемент вказує на фрейм фізичної пам'яті. Фізичну адресу каталогу сторінок зберігають у блоці KPROCESS.

Розмір лінійної адреси, з якою працює система, становить 32 біти. З них 10 біт відповідають адресі в каталозі сторінок, ще 10 – це індекс елемента в таблиці, останні 12 біт адресують конкретний байт сторінки (і є зсувом).

Розмір елемента таблиці сторінок теж становить 32 біти. Перші 20 біт адресують конкретний фрейм (і використовуються разом із останніми 12 біт лінійної адреси), а інші 12 біт описують атрибути сторінки (захист, стан сторінки в пам'яті, який файл підкачування використовує). Якщо сторінка не перебуває у пам'яті, то в перших 20 біт зберігають зсув у файлі підкачування.

Для платформо-незалежного визначення розміру сторінки у Win32 API використовують універсальну функцію отримання інформації про систему GetSystemInfo():

```
SYSTEM_INFO info;   // структура для отримання інформації про систему
GetSystemInfo(&info);
printf("Розмір сторінки: %d\n", info.dwPageSize);
```

8.6.3. Особливості адресації процесів і ядра

Лінійний адресний простір процесу поділяється на дві частини: перші 2 Гбайт адрес доступні для процесу в режимі користувача і є його захищеним адресним простором; інші 2 Гбайт адрес доступні тільки в режимі ядра і відображають системний адресний простір.

Зазначимо, що таке співвідношення між адресним простором процесу і ядра відрізняється від прийнятого в Linux (3 Гбайт для процесу, 1 Гбайт для ядра).

Деякі версії Windows XP дають можливість задати співвідношення 3 Гбайт/1 Гбайт під час завантаження системи.

8.6.4. Структура адресного простору процесів і ядра

В адресному просторі процесу можна виділити такі ділянки:

- перші 64 Кбайт (починаючи з нульової адреси) – це спеціальна ділянка, доступ до якої завжди спричиняє помилки;
- усю пам'ять між першими 64 Кбайт і останніми 136 Кбайт (майже 2 Гбайт) може використовувати процес під час свого виконання;
- далі розташовані два блоки по 4 Кбайт: блоки оточення потоку (TEB) і процесу (PEB) (див. розділ 3);
- наступні 4 Кбайт – ділянка пам'яті, куди відображаються різні системні дані (системний час, значення лічильника системних годин, номер версії системи), тому для доступу до них процесу не потрібно перемикатися в режим ядра;
- останні 64 Кбайт використовують для запобігання спробам доступу за межі адресного простору процесу (спроба доступу до цієї пам'яті дасть помилку).

Системний адресний простір містить велику кількість різних ділянок. Найважливіші з них наведено нижче.

- Перші 512 Мбайт системного адресного простору використовують для завантаження ядра системи.
- 4 Мбайт пам'яті виділяють під каталог сторінок і таблиці сторінок процесу.
- Спеціальну ділянку пам'яті розміром 4 Мбайт, яку називають гіперпростором (hyperspace), використовують для відображення різних структур даних, специфічних для процесу, на системний адресний простір (наприклад, вона містить список сторінок робочого набору процесу).
- 512 Мбайт виділяють під системний кеш.
- У системний адресний простір відображаються спеціальні ділянки пам'яті – вивантажуваний пул і невивантажуваний пул, які розглянемо в розділі 10.
- Приблизно 4 Мбайт у самому кінці системного адресного простору виділяють під структури даних, необхідні для створення аварійного образу пам'яті, а також для структур даних HAL.

Висновки

- Керування пам'яттю є однією з найскладніших задач, які стоять перед операційною системою. Щоб нестача пам'яті не заважала роботі користувача, потрібно розв'язувати задачу координації різних видів пам'яті. Можна використовувати повільнішу пам'ять для збільшення розміру швидшої (на цьому ґрунтується технологія віртуальної пам'яті), а також швидшу – для прискорення доступу до повільнішої (на цьому ґрунтується кешування).
- Технологія віртуальної пам'яті передбачає введення додаткових перетворень між логічними адресами, які використовують програми, та фізичними, що їх

розуміє мікросхема пам'яті. На основі таких перетворень може бути реалізований захист пам'яті; крім того, вони дають змогу процесу розміщатися у фізичній пам'яті не неперервно і не повністю (ті його частини, які в цей момент часу не потрібні, можуть бути збережені на жорсткому диску). Ця технологія спирається на той факт, що тільки частина адрес процесу використовується в конкретний момент часу, тому коли зберігати в основній пам'яті тільки її, продуктивність процесу залишиться прийнятною.

- Базовими підходами до реалізації віртуальної пам'яті є сегментація і сторінкова організація. Обидва ці підходи дають можливість розглядати логічний адресний простір процесу як сукупність окремих блоків, кожен з яких може бути відображений на основну пам'ять або на диск. Головна відмінність полягає в тому, що у випадку сегментації блоки мають змінну довжину, а у разі сторінкової організації – постійну. Сьогодні часто трапляється комбінація цих двох підходів (сторінково-сегментна організація пам'яті).
- У разі сторінкової організації пам'яті логічна адреса містить номер у спеціальній структурі даних – таблиці сторінок, а також зсув відносно початку сторінки. Розділення адреси на частини відбувається апаратно. Елемент таблиці сторінок містить адресу початку блоку фізичної пам'яті, у який відображається сторінка (такий блок називають фреймом) і права доступу. Він може також відповідати сторінці, відображеній на диск. Таблиці сторінок можуть містити кілька рівнів. Таблицю верхнього рівня називають каталогом сторінок. Кожний процес має свій набір таких таблиць. Для прискорення доступу останні використані елементи таблиць сторінок кешуються в асоціативній пам'яті.

Контрольні запитання та завдання

1. Чи є необхідність реалізувати в системі віртуальну пам'ять, якщо відомо, що загальний обсяг пам'яті, необхідної для всіх активних процесів, ніколи не перевищить обсяг доступної фізичної пам'яті? Якщо така необхідність є, то які функції системи віртуальної пам'яті варто реалізувати обов'язково, а які – ні?
2. Перелічіть відмінності в реалізації та використанні сегментів даних і сегментів коду.
3. У якій ситуації виконання двох незалежних процесів, що коректно завершуються в системі зі сторінковою організацією пам'яті, призведе до взаємного блокування в системі, де такої організації немає? Передбачено, що запит на виділення пам'яті переводить процес у режим очікування, якщо для його виконання бракує фізичної пам'яті.
4. Чому розмір сторінки повинен бути степенем числа 2?
5. Припустімо, що розмір сторінки пам'яті становить 4 Кбайт, кожен її елемент займає 4 байти. Кожна таблиця сторінок повинна вміщатися на одній сторінці. Скільки рівнів таблиць сторінок буде потрібно, щоб адресувати:
   а) 32-бітний адресний простір;
   б) 64-бітний адресний простір?

6. Коли використання звичайного масиву з лінійним пошуком виявляється неефективним для реалізації таблиці сторінок? Які поліпшення в цьому випадку можна запропонувати?
7. Перелічіть можливі переваги сторінково-сегментної організації пам'яті порівняно з чистою сегментацією і чисто сторінковою організацією.
8. Поясніть, чому в разі використання сторінково-сегментної організації пам'яті перетворення адреси не може бути реалізоване у зворотному порядку (коли логічну адресу спочатку перетворюють за допомогою таблиці сторінок, а потім — за допомогою таблиці сегментів).
9. У системі зі сторінково-сегментною організацією пам'яті кожному процесові виділяють 64 Кбайт адресного простору для трьох сегментів: коду, даних і стека. Розмір сегмента коду для процесу дорівнює 32 Кбайт, сегмента даних — 16 400 байт, а стека — 15 800 байт. Чи достатньо адресного простору процесу для розміщення цих сегментів, якщо розмір сторінки дорівнює:
   а) 4 Кбайт;
   б) 512 байт?
10. Чи можуть під час виконання програми всі сегментні регістри містити однакові значення? Навіщо це може знадобитися?

Розділ 9

Взаємодія з диском під час керування пам'яттю

- Підкачування сторінок пам'яті
- Зберігання і заміщення сторінок пам'яті
- Резидентна множина і пробуксовування
- Організація віртуальної пам'яті в Linux та Windows XP

Дотепер ми розглядали загальні питання реалізації перетворення адрес, на якому ґрунтується технологія віртуальної пам'яті. Як зазначалося, ця технологія дає можливість відображати на основну пам'ять тільки ту частину адресного простору процесу, яку в конкретний момент він активно використовує, а решту — на диск. У цьому розділі йтиметься про особливості реалізації такого відображення.

9.1. Причини використання диска під час керування пам'яттю

Тимчасове збереження окремих частин адресного простору на диску допомагає розв'язати одну з основних проблем, що виникають під час реалізації керування пам'яттю в ОС, а саме: організацію завантаження і виконання програм, які окремо або разом не вміщаються в основній пам'яті.

Найпростішим і найдавнішим підходом є завантаження і вивантаження всього адресного простору процесу за один прийом. Процеси завантажуються у пам'ять повністю, виконуються певний час, а потім так само повністю вивантажуються на диск. Отже, процес або весь перебуває у пам'яті, або цілком зберігається на диску (про такий процес прийнято говорити, що він перебуває у вивантаженому стані). Така технологія має низку недоліків:

- її використання призводить до значної фрагментації зовнішньої пам'яті;
- вона не дає змоги виконувати процеси, які мають потребу у більшому обсязі пам'яті, ніж доступно у системі;
- погано підтримуються процеси, які можуть виділяти собі додаткову динамічну пам'ять (це необхідно робити з урахуванням можливого розширення адресного простору процесу).

Вивантаження всього процесу із пам'яті у сучасних ОС можна використовувати як засіб зниження навантаження, але лише на доповнення до інших технологій взаємодії з диском.

9.2. Поняття підкачування

Описана технологія повного завантаження і вивантаження процесів традиційно називалася підкачуванням або *простим підкачуванням* [44], але тут вживатимемо цей термін у ширшому значенні. У сучасних ОС під *підкачуванням* (swapping) розуміють увесь набір технологій, які здійснюють взаємодію із диском під час реалізації віртуальної пам'яті, щоб дати можливість кожному процесу звертатися до великого діапазону логічних адрес за рахунок використання дискового простору.

Розглянемо загальні принципи підкачування. Як відомо, зняття вимоги неперервності фізичного простору, куди відображається адресний простір процесу, і можливість переміщення процесу в пам'яті під час його виконання дає змогу не тримати одночасно в основній пам'яті всі блоки пам'яті (сторінки або сегменти), які утворюють адресний простір цього процесу. Під час завантаження процесу в основну пам'ять у ній розміщають лише кілька його блоків, які потрібні для початку роботи. Частину адресного простору процесу, що у конкретний момент часу відображається на основну пам'ять, називають *резидентною множиною* процесу (resident set). Поки процес звертається тільки до пам'яті резидентної множини, виконання процесу не переривають. Як тільки здійснюється посилання на блок, що перебуває за межами резидентної множини (тобто відображений на диск), відбувається апаратне переривання. Оброблювач цього переривання призупиняє процес і запускає дискову операцію читання потрібного блоку із диска в основну пам'ять. Коли блок зчитаний, операційну систему сповіщають про це, після чого процес переводять у стан готовності й, зрештою, поновлюють, після чого він продовжує свою роботу, ніби нічого й не сталося; на момент його поновлення потрібний блок уже перебуває в основній пам'яті, де процес і розраховував його знайти.

Реалізація підкачування використовує правило «дев'яносто до десяти». Ідеальною реалізацією керування пам'яттю є надання кожному процесові пам'яті, за розміром порівнянної із жорстким диском, а за швидкістю доступу — з основною пам'яттю. Оскільки за правилом «дев'яносто до десяти» на 10 % адресного простору припадає 90 % посилань на пам'ять, як деяке наближення до ідеальної реалізації можна розглядати такий підхід: зберігати ці 10 % в основній пам'яті, а інший адресний простір відображати на диск. Як показано на рис. 9.1, частіше використовують сторінки 0, 3, 4, 6, тому їхній вміст зберігають в основній пам'яті, а сторінки 1, 2, 5, 7 використовують рідше, тому їхній вміст зберігають на диску.

Головною проблемою залишається ухвалення рішення про те, які із блоків пам'яті мають в конкретний момент відображатися на основну пам'ять, а які — на диск.

Внаслідок використання технології підкачування кількість виконуваних процесів збільшується (для кожного з них в основній пам'яті перебуватиме тільки частина блоків). Підкачування дає також змогу виконувати процеси, які за розміром більші, ніж основна пам'ять (для таких процесів у різні моменти часу в основну пам'ять відображатимуться різні блоки).

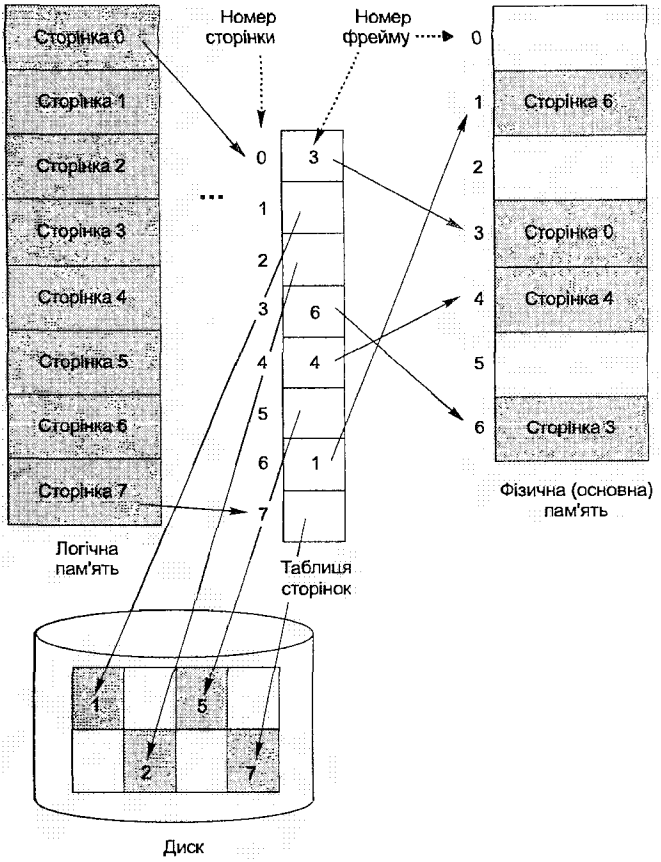

Рис. 9.1. Принцип дії підкачування

У цьому розділі йтиметься про підкачування, яке використовується у поєднанні зі сторінковою організацією пам'яті. Слід зазначити, що підкачування аналогічним чином може бути реалізоване й на основі сегментації, хоча ця задача виглядає складнішою через те, що наперед не відомо, якого розміру блоки потрібно буде зберігати на диску.

9.3. Завантаження сторінок на вимогу. Особливості підкачування сторінок

Базовий підхід, який використовують під час реалізації підкачування сторінок із диска, називають технологією *завантаження сторінок на вимогу* (demand paging).

Ця технологія діє у припущенні, що не всі сторінки процесу мають завантажуватися у пам'ять негайно. Завантажуються тільки ті, що необхідні для початку його роботи, інші — коли стають потрібні. Процес переносу сторінки із диска у пам'ять називають *завантаженням із диска* (swapping in), процес переносу сторінки із пам'яті на диск — *вивантаженням на диск* (swapping out).

9.3.1. Апаратна підтримка підкачування сторінок

Для організації апаратної підтримки підкачування кожний елемент таблиці сторінок має містити спеціальний біт — біт присутності сторінки у пам'яті P. Коли він дорівнює одиниці, то це означає, що відповідна сторінка завантажена в основну пам'ять, і їй відповідає деякий фрейм. Якщо біт присутності сторінки дорівнює нулю, це означає, що дана сторінка перебуває на диску і має бути завантажена в основну пам'ять перед використанням (рис. 9.2). У таблиці сторінок архітектури IA-32 (див. розділ 8.3.4) даний біт називають Present.

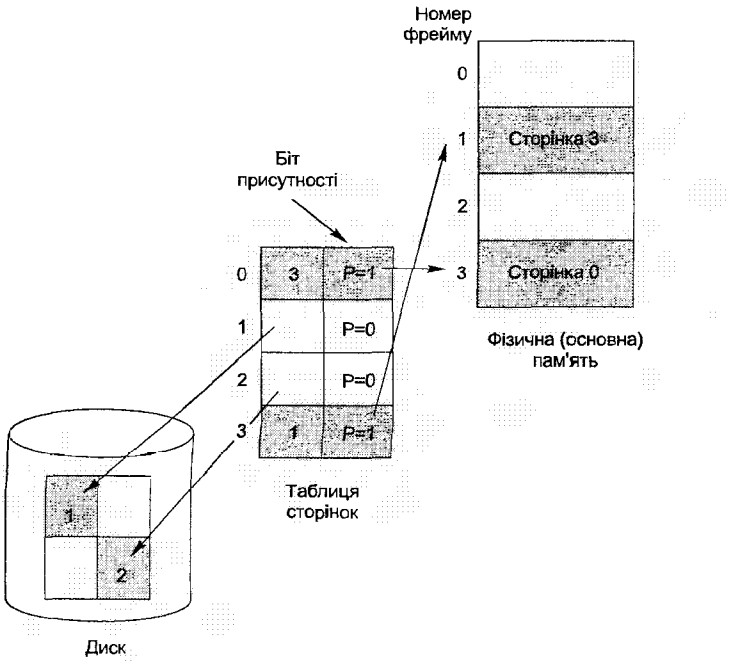

Рис. 9.2. Біт присутності сторінки

Для сторінки може бути заданий *біт модифікації M*. Його спочатку (під час завантаження сторінки із диска) покладають рівним нулю, потім — одиниці, якщо сторінку модифікують, коли вона завантажена у фрейм основної пам'яті. Наявність такого біта дає змогу оптимізувати операції вивантаження сторінок на диск. Коли намагаються вивантажити на диск вміст сторінки, для якої біт *M* дорівнює нулю, це означає, що записування на диск можна проігнорувати, бо вміст сторінки не змінився від часу завантаження у пам'ять. У розділі 9.5 побачимо, як цей біт можна використати для реалізації заміщення сторінок (визначення сторінки, яку потрібно вивантажити на диск у разі нестачі фреймів фізичної пам'яті). У таблиці сторінок архітектури IA-32 даний біт називають Dirty.

Якщо процес працює тільки зі сторінками, для яких біт *P* дорівнює одиниці (його резидентна множина не змінюється), складається враження, що всі його сторінки є у пам'яті, хоча насправді це може бути і не так (просто сторінок, яких немає у пам'яті, у цьому разі взагалі не використовують).

9.3.2. Поняття сторінкового переривання

Коли процес робить спробу доступу до сторінки, для якої біт P дорівнює нулю, то, як ми вже бачили під час вивчення загальної стратегії підкачування, відбувається апаратне переривання. Його називають *сторінковим перериванням* або *сторінковою відмовою* (page fault). ОС має обробити це переривання.

- Сторінку перевіряють на доступність для цього процесу (діапазон доступної пам'яті зберігається у структурі даних процесу).
- Якщо сторінка недоступна, процес переривають, якщо доступна, знаходять вільний фрейм фізичної пам'яті та ставлять у чергу дискову операцію читання потрібної сторінки в цей фрейм.
- Після читання модифікують відповідний запис таблиці сторінок (біт присутності покладають рівним одиниці).
- Перезапускають інструкцію, що викликала апаратне переривання.

Тепер процес може отримати доступ до сторінки, ніби вона завжди була в пам'яті. Процес поновлюють у тому самому стані, в якому він був перед перериванням.

Чисте завантаження сторінок на вимогу зводиться до того, що жодну сторінку не завантажують у пам'ять завчасно. При цьому після запуску процесу жодна з його сторінок не перебуватиме в пам'яті: усі вони будуть довантажуватися внаслідок сторінкових переривань за потребою. Є альтернативні підходи (наприклад, попереднє завантаження сторінок), які будуть описані далі.

Зазначимо, що в загальному випадку сторінкове переривання не обов'язково спричиняє підкачування сторінки з диска; воно може бути результатом звертання до сторінки, що не належить до адресного простору процесу. У цьому разі має бути згенерована помилка. З іншого боку, причиною сторінкового переривання під час чистого завантаження на вимогу може бути звертання до зовсім нової сторінки пам'яті, яка жодного разу не була використана процесом. Обробляч може зарезервувати новий фрейм, проініціалізувати його (наприклад, заповнити нулями) і поставити йому у відповідність сторінку.

9.3.3. Продуктивність завантаження на вимогу

Реалізація завантаження на вимогу із підкачуванням сторінок із диска може істотно впливати на продуктивність ОС. Зробимо найпростіший розрахунок такого впливу.

Спрощену характеристику, яку використовують для оцінки продуктивності системи із завантаженням сторінок на вимогу, називають *ефективним часом доступу* і розраховують за такою формулою:

$$T_{ea} = (1 - P_{pf}) \times T_{ma} + P_{pf} \times T_{pf},$$

де P_{pf} — імовірність сторінкового переривання; T_{ma} — середній час доступу до пам'яті; T_{pf} — середній час обробки сторінкового переривання.

У сучасних системах час доступу до пам'яті T_{ma} становить від 20 до 200 нс. Розрахуємо час обробки сторінкового переривання T_{pf}.

Коли відбувається сторінкове переривання, система виконує багато різноманітних дій (генерацію переривання, виклик оброблювача, збереження регістрів процесора, перевірку коректності доступу до пам'яті, початок операції читання із диска, перемикання контексту на інший процес, обробку переривання від диска про завершення операції читання, корекцію таблиці сторінок тощо). У результаті кількість інструкцій процесора, необхідних для обробки сторінкового переривання, виявляється дуже великою (десятки тисяч), і середній час обробки сягає *кількох мілісекунд* (одна мілісекунда відповідає мільйону наносекунд).

Припустимо, що величина T_{ma} становить 100 нс, T_{pf} – 10 мс, а ймовірність виникнення сторінкового переривання – 0,001 (одне переривання на 1000 звертань до пам'яті, цю величину називають *рівнем сторінкових переривань* – page fault rate). Тоді ефективний час доступу буде рівний:

$$T_{ea} = (1 - 0{,}001) \times 100 + 0{,}001 \times 10\,000\,000 = 10\,099{,}9 \text{ нс.}$$

Як бачимо, внаслідок сторінкових переривань, що виникають з імовірністю одна тисячна, ефективний час доступу виявився в 100 разів більшим, ніж під час роботи з основною пам'яттю, тобто продуктивність системи знизилася в 100 разів. Щоб здобути зниження продуктивності на прийнятну величину, наприклад на 10 %, необхідно, щоб імовірність сторінкового переривання була приблизно одна мільйонна:

$$P_{pf} = (0{,}1 \times T_{ma}) / (T_{pf} - T_{ma}) = 10 / 9\,999\,900 = 1 / 999\,990.$$

Для досягнення прийнятної продуктивності системи із завантаженням сторінок на вимогу рівень сторінкових переривань має бути надзвичайно низьким, тому будь-які поліпшення в алгоритмах керування пам'яттю, що знижують цей рівень, завжди виправдані. По суті, будь-які витрати часу на виконання найскладніших алгоритмів окупатимуться, якщо зменшується кількість сторінкових переривань, оскільки одне таке переривання обробляється повільніше, ніж виконується алгоритм майже будь-якої складності.

9.4. Проблеми реалізації підкачування сторінок

Під час реалізації підкачування потрібно дати відповідь на такі запитання.

1. Які сторінки потрібно завантажити із диска і в який час?
   Відповідь на це запитання визначає прийнята в цій системі *стратегія вибірки сторінок*. Однією з таких стратегій є завантаження сторінок на вимогу, яку щойно було розглянуто, коли сторінку завантажують у пам'ять тільки під час обробки сторінкового переривання. Іншою стратегією такого роду є *попереднє завантаження сторінок* (prepaging), коли у пам'ять завантажують кілька сторінок, розташованих на диску послідовно для того щоб зменшити кількість таких сторінкових переривань. Ця технологія ґрунтується на принципі локалізації, але переваги у продуктивності у разі її застосування довести не вдалося.
2. Яку сторінку потрібно вивантажити на диск, коли вільних фреймів у основній пам'яті більше немає?

Відповідь на це запитання визначає *стратегія заміщення сторінок*. Різні підходи до реалізації цієї стратегії буде розглянуто в розділі 9.5.

3. Де у фізичній пам'яті мають розміщуватися сторінки процесу?
Відповідь на це запитання визначає *стратегія розміщення*. Вона відіграє важливу роль під час реалізації підкачування на основі сегментації, у разі підкачування на основі сторінкової організації вона не така важлива, оскільки всі сторінки рівноправні й можуть перебувати у пам'яті де завгодно.
4. Яким чином організувати керування резидентною множиною?
Головне завдання такого керування — забезпечити, щоб у резидентній множині були тільки сторінки, які дійсно потрібні процесу. Про особливості керування резидентною множиною розкажемо у розділі 9.7.

9.5. Заміщення сторінок

Під час генерації сторінкового переривання може виникнути ситуація, коли жодного вільного фрейму у фізичній пам'яті немає. Причини появи цієї проблеми полягають в особливостях використання завантаження сторінок на вимогу.

Припустимо, що в нашій системі кожен процес у конкретний момент часу в середньому звертається тільки до половини своїх сторінок. Завантаження сторінок на вимогу залишає половину сторінок кожного процесу на диску, вивільняючи половину фреймів фізичної пам'яті, які міг би зайняти цей процес, для виконання інших процесів. Тому можна завантажити у пам'ять більше процесів.

Нехай основна пам'ять містить 60 фреймів. Якщо не використовувати завантаження на вимогу, в основну пам'ять можна завантажити шість процесів, кожен із яких використає 10 сторінок, причому кожній сторінці буде відповідати певний фрейм. У разі використання завантаження на вимогу ці шість процесів у конкретний момент часу використовуватимуть тільки 30 фреймів пам'яті, а інші 30 залишатимуться вільними, тому можна завантажити в них ще до шести процесів (у сумі — до дванадцяти).

Припустимо, що у пам'ять завантажено одночасно вісім процесів; у цьому разі процесор використовуватиметься активніше, ніж для шести запущених процесів, крім того, вивільниться 20 фреймів. Однак загальний обсяг адресного простору завантажених процесів перевищуватиме обсяг фізичної пам'яті.

Поки процеси використовують свої сторінки звичайним чином (на 50 %), всі вони будуть виконуватися в основній пам'яті без проблем. Але завжди може виникнути ситуація, коли процеси зажадають більше пам'яті, ніж їм це потрібно в середньому. Згадані вісім процесів можуть у якийсь момент зажадати весь свій адресний простір, тому всього їм знадобиться 80 фреймів основної пам'яті (а їх тільки 60). Що більше процесів одночасно виконується в системі, то вища ймовірність виникнення такої ситуації. Як домогтися того, щоб вона виникала не дуже часто, розглянемо в розділі 9.7. У цьому розділі зупинимося на діях, яких вимагають від ОС, коли така ситуація вже виникла.

За відсутності вільного фрейму в пам'яті ОС має вибрати, яку сторінку вилучити із пам'яті для того щоб вивільнити місце для нової сторінки. Процес цього вибору визначає *стратегія заміщення сторінок*. Коли сторінка, яку вилучають,

була змінена, необхідно вивантажити її на диск для відображення цих змін, коли ж вона залишилася незмінною (наприклад, це була сторінка із програмним кодом), цього робити не потрібно. Для індикації того, чи була сторінка змінена, використовують біт модифікації сторінки *M* (про нього йшлося в розділі 9.3.1).

Яку саме сторінку потрібно вилучати із пам'яті, визначає *алгоритм заміщення сторінок* (page replacement algorithm). Вибір такого алгоритму відчутно впливає на продуктивність системи, тому що вдало підібраний алгоритм зменшує кількість сторінкових переривань, а невдала його реалізація може її значно збільшити (якщо вилучити часто використовувану сторінку, то вона незабаром може знову знадобитися і т. д.). Навіть невеликі поліпшення в методах заміщення сторінок можуть спричинити великий загальний виграш у продуктивності.

Слід зазначити, що алгоритми заміщення сторінок (особливо ті, які справді дають виграш у продуктивності) досить складні в реалізації і майже завжди потребують апаратної підтримки (хоча б у вигляді наявності біта модифікації сторінки *M*).

Ось який вигляд матиме алгоритм завантаження сторінки з урахуванням заміщення сторінок.

1. Знайти вільний фрейм у фізичній пам'яті:
   а) якщо вільний фрейм є, використати його (перейти до кроку 2);
   б) якщо вільного фрейму немає, використати алгоритм заміщення сторінок для того щоб знайти *фрейм-жертву* (victim frame);
   в) записати фрейм-жертву на диск (якщо біт модифікації для нього ненульовий), відповідно змінити таблицю сторінок і таблицю вільних фреймів.
2. Знайти потрібну сторінку на диску.
3. Прочитати потрібну сторінку у вільний фрейм (якщо раніше були виконані кроки *б* і *в* п. 1, цим фреймом буде той, котрий щойно вивільнився).
4. Знову запустити інструкцію, на якій відбулося переривання.

9.5.1. Оцінка алгоритмів заміщення сторінок. Рядок посилань

Оскільки реалізація алгоритмів заміщення сторінок важлива з погляду продуктивності системи, вони ретельно досліджувалися, були виділені критерії їхнього порівняння і формат даних для тестування.

Критерієм для порівняння алгоритмів заміщення звичайно є рівень сторінкових переривань: що він нижчий, то кращим вважають алгоритм. Оцінку алгоритму здійснюють через підрахунок кількості сторінкових переривань, які виникли внаслідок його запуску для конкретного набору посилань на сторінки у пам'яті. Набір посилань на сторінки подають *рядком посилань* (reference string) із номерами сторінок. Такий рядок можливо згенерувати випадково, а можна отримати відстеженням посилань на пам'ять у реальній системі (у цьому разі даних може бути зібрано дуже багато). Групу із кількох посилань, що йдуть підряд, на одну й ту саму сторінку в рядку відображають одним номером сторінки, оскільки такий набір посилань не може спричинити сторінкового переривання.

У цьому розділі використовуватимемо такий рядок посилань:

1, 2, 3, 5, 2, 4, 2, 4, 3, 1, 4.

Щоб оцінити алгоритми заміщення, потрібно також зазначити кількість доступних фреймів. Чим вона більша, тим меншим буде рівень сторінкових переривань, тому алгоритми оцінюють для однакових її значень. Зазначимо, що, коли доступний один фрейм, кожен елемент рядка посилань викликатиме сторінкове переривання; з іншого боку, якщо кількість доступних фреймів дорівнює кількості різних сторінок у рядку посилань, сторінкових переривань не буде зовсім. Алгоритми оцінюватимемо для трьох доступних фреймів.

9.5.2. Алгоритм FIFO

Найпростішим у реалізації (за винятком алгоритму випадкового заміщення) є *алгоритм FIFO*. Він дозволяє замінити сторінку, яка перебувала у пам'яті найдовше.

На рис. 9.3 зображена робота цього алгоритму на рядку посилань (кольором тут і далі виділено сторінкові переривання).

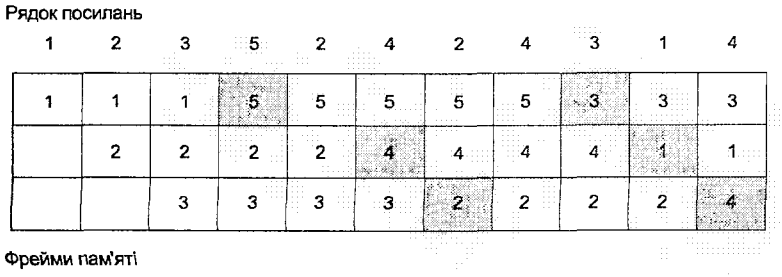

Рис. 9.3. FIFO-алгоритм заміщення сторінок

Для реалізації такого алгоритму досить підтримувати у пам'яті список усіх сторінок, організований за принципом FIFO-черги (звідси й назва алгоритму). Коли сторінку завантажують у пам'ять, її додають у хвіст черги, у разі заміщення її вилучають з голови черги.

Основними перевагами цього алгоритму є те, що він не потребує апаратної підтримки (тому для архітектур, які такої підтримки не надають, він може бути *єдиним варіантом реалізації заміщення сторінок*).

Головним недоліком алгоритму FIFO є те, що він не враховує інформації про використання сторінки. Такий алгоритм може вибрати для вилучення, наприклад, сторінку із важливою змінною, котра вперше отримала своє значення на початку роботи, і з того часу її постійно використовують та модифікують. Вилучення такої сторінки із пам'яті призводить до того, що система буде негайно змушена знову звернутися по неї на диск.

Такий алгоритм недостатньо ефективний, і його варто використовувати тоді, коли кращого алгоритму реалізувати не вдається (або у поєднанні з іншими підходами, наприклад, з буферизацією сторінок, яку буде описано в розділі 9.5.6). Деякі інші, ефективніші алгоритми в найгіршому випадку можуть зводитися до алгоритму FIFO.

Наголосимо, однак, що жоден алгоритм заміщення не може призвести до некоректної роботи програми. Після того як сторінка буде завантажена у пам'ять із диска, робота з нею відбуватиметься так, ніби вона була в пам'яті від самого початку. Різниця полягатиме тільки у продуктивності (втім, ця різниця може виявитися дуже істотною).

9.5.3. Оптимальний алгоритм

Є алгоритм заміщення сторінок, оптимальність якого теоретично доведена (тобто він буде гарантовано кращим за будь-який інший алгоритм). Він зводиться до таких дій: замінити сторінку, яку не використовуватимуть найдовше.

Приклад використання такого алгоритму зображено на рис. 9.4.

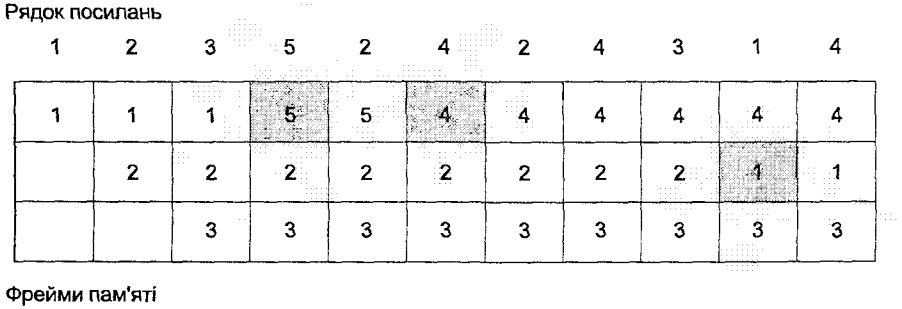

Рис. 9.4. Оптимальний алгоритм заміщення сторінок

На жаль, у загальному випадку реалізувати оптимальний алгоритм заміщення сторінок неможливо, бо він вимагає знання того, як у майбутньому буде поводитися процес (цим він схожий на інший теоретично оптимальний алгоритм – алгоритм планування STCF, описаний у розділі 4).

З іншого боку, якщо є конкретний набір сторінок для процесу, можна його запустити і зібрати інформацію про поведінку кожної сторінки; під час наступних запусків можна заміщувати сторінки оптимально. Це може бути корисно для оцінки продуктивності алгоритмів заміщення (під час тестування алгоритму завжди корисно знати, наскільки він гірший від оптимального для конкретних умов).

9.5.4. Алгоритм LRU

Оскільки оптимальний алгоритм заміщення сторінок прямо реалізувати неможливо, основним завданням розробників має бути максимальне наближення до оптимального алгоритму. Опишемо основні принципи такого наближення.

Головною особливістю оптимального алгоритму (крім використання знання про майбутнє, що на практиці реалізувати не можна) є те, що він ґрунтується на збереженні для кожної сторінки інформації про те, коли до неї зверталися востаннє. Збереження цієї інформації за умови заміни майбутнього часу на минулий привело до найефективнішого алгоритму з тих, які можна реалізувати – алгоритму *LRU* (Least Recently Used – алгоритм заміщення сторінки, не використовуваної найдовше). Формулюють його так: замінити сторінку, що не була використана упродовж найбільшого проміжку часу.

Рис. 9.5 ілюструє роботу цього алгоритму. По суті, LRU – це оптимальний алгоритм, розгорнутий за часом назад, а не вперед.

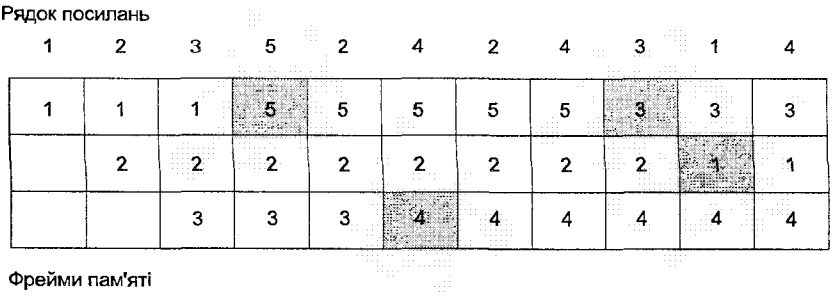

Рис. 9.5. LRU-алгоритм заміщення сторінок

Основні труднощі під час використання LRU-алгоритму полягають у тому, що його складно реалізувати, оскільки потрібно зберігати інформацію про кожне звертання до пам'яті так, щоб не страждала продуктивність. Потрібна набагато серйозніша апаратна підтримка, ніж наявність біта модифікації або асоціативна пам'ять. Таку підтримку можуть забезпечувати тільки деякі спеціалізовані архітектури. Розглянемо деякі можливі варіанти реалізації LRU-алгоритму.

- Можна організувати всі номери сторінок у вигляді двозв'язного списку. Під час кожного звертання до сторінки її вилучають зі списку (можливо, із середини) і поміщають у його початок. Тому сторінка, до якої зверталися найпізніше, буде завжди на початку списку, а та, до якої не зверталися найдовше (тобто жертва), – позаду.
- Можна організувати в процесорі глобальний лічильник (завдовжки, наприклад, 64 біти), збільшувати його на кожній інструкції та зберігати у відповідному елементі таблиці сторінок у разі звертання до кожної сторінки. Тоді потрібно замінювати сторінку із найменшим значенням лічильника.

Основний недолік цих підходів – низька продуктивність. Наприклад, якщо для поновлення лічильника або стека організовувати переривання, то обробляч цього переривання буде виконуватися після кожної інструкції доступу до пам'яті, сповільнюючи доступ до пам'яті в кілька разів.

9.5.5. Годинниковий алгоритм

Базовий годинниковий алгоритм

Хоч алгоритм LRU реалізувати дуже важко і його апаратна підтримка є лише в деяких системах, все-таки сучасні апаратні архітектури дають змогу хоча б частково використати закладену в ньому ідею – виконати його наближення. Насамперед вони підтримують *біт використання сторінки* (reference bit, *R*), що перебуває в елементі таблиці сторінок і стає рівним одиниці у разі звертання до відповідної сторінки. Наявність такого біта дає змогу з'ясувати факт звертання до сторінки (не даючи змоги, однак, упорядкувати сторінки за часом звертання до них, що необхідно для LRU-алгоритму). Як було вже показано (розділ 8.3.4), в архітектурі IA-32 біт *R* називають прапорцем доступу (Accessed).

Наявність біта використання сторінки є основою *алгоритму другого шансу*, або *годинникового алгоритму* (clock algorithm), – одного з найефективніших реально застосовуваних алгоритмів.

Опишемо цей алгоритм. Передусім сторінки для нього мають бути організовані в кільцевий список (їх можна зобразити у вигляді циферблата годинника). Покажчик, який використовують під час сканування списку сторінок, ще називають *стрілкою* (подібно до стрілки годинника).

Спочатку беруть сторінку, що найдовше перебуває у пам'яті (як для FIFO). Якщо її біт використання (R) дорівнює 0, то сторінку негайно замінюють, поміщаючи на її місце нову. Якщо ж біт R дорівнює 1 (до сторінки зверталися), то його покладають рівним 0 (начебто ця сторінка тільки що завантажена у пам'ять), і прохід за списком триває далі, поки не буде знайдена сторінка з $R = 0$ (а доти біт R для всіх сторінок покладають рівним 0). Знайдену сторінку замінюють, після чого для нової сторінки задають $R = 1$ і наставляють на неї стрілку (рис. 9.6). У найгіршому випадку (якщо для всіх сторінок біт R дорівнює одиниці) почнеться друге коло обходу списку (другий шанс) і буде замінена найстарша сторінка із тих, що були пройдені на першому колі. У цьому разі алгоритм зводиться до алгоритму FIFO.

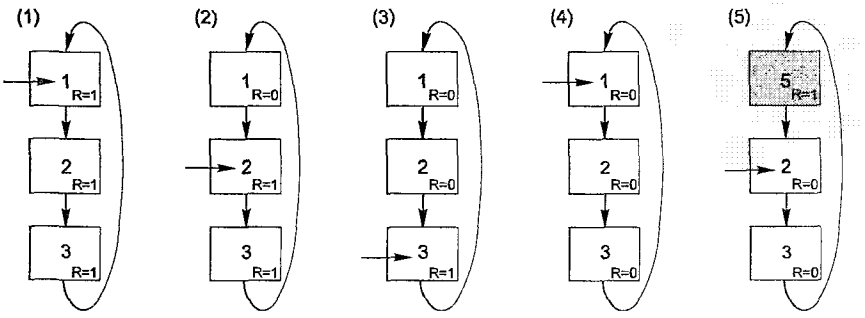

Рис. 9.6. Годинниковий алгоритм заміщення сторінок

На рис. 9.7 зображено результат застосування годинникового алгоритму для попереднього рядка посилань; на кожному кроці показане поточне положення стрілки і значення R для кожного фрейму. Зазначимо, що для цього рядка і трьох фреймів годинниковий алгоритм повністю ідентичний LRU (на практиці так буває не завжди).

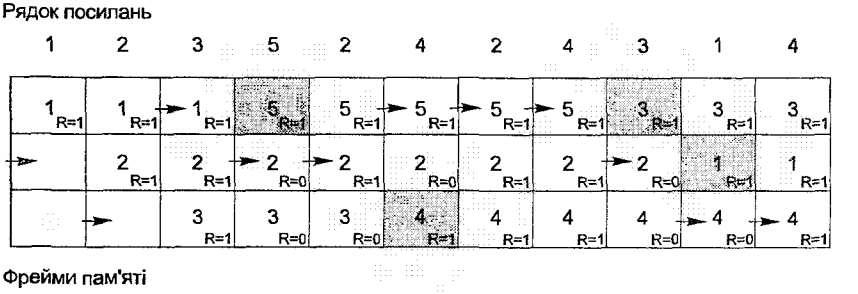

Рис. 9.7. Результат застосування годинникового алгоритму заміщення сторінок

Годинниковий алгоритм із додатковими бітами

Проблемою годинникового алгоритму є його підвищена чутливість до інтервалу часу між обходами стрілки. Якщо обхід відбувається надто рідко, виникає ситуація, коли всі або майже всі сторінки матимуть $R = 1$ і доводиться заходити на друге коло. Якщо ж обхід відбуватиметься надто часто, втрачатиметься інформація про використання сторінок (вона постійно затиратиметься стрілкою). Для вирішення цієї проблеми був запропонований *годинниковий алгоритм із додатковими бітами.*

У разі використання цього алгоритму кожну сторінку супроводжують не один, а n бітів, які є лічильником використання сторінки C_u (наприклад, при n=8 лічильник можна розглядати як ціле завдовжки один байт), а також біт R. Під час використання сторінки біт R покладають рівним 1. Час від часу (за перериванням від таймера) ОС обходить циклічний список сторінок і поміщає значення біта R у старший біт лічильника, а інші біти зсуває вправо на 1 біт, при цьому молодший біт відкидають. Формула перерахування має такий вигляд:

$$C_u = (\mathrm{R} \ll (n-1)) \mid (C_u \gg 1)$$

Біти лічильника в цьому випадку містять історію використання сторінки впродовж останніх n періодів часу між обходами списку. Наприклад, якщо сторінка за цей час не була використана жодного разу, всі біти C_u для неї будуть рівні 0, якщо її використовували щоразу — одиниці. Кожне звертання до сторінки робить значення лічильника використання для неї більшим, ніж для всіх сторінок, які на цьому інтервалі не були використані. Наприклад, бітове значення 10000000 = 128 (сторінка щойно була використана) буде більшим за 01111111 = 127 (сторінка була використана на всіх попередніх інтервалах, крім останнього).

За необхідності вибору сторінки для заміщення береться сторінка із найменшим значенням лічильника C_u. Така сторінка буде LRU-сторінкою для останніх n періодів часу між обходами списку.

Якщо сторінок із найменшим значенням лічильника декілька, можна вилучити із пам'яті їх усі або використати FIFO-алгоритм для вибору однієї з них.

9.5.6. Буферизація сторінок

Ще однією важливою технологією заміщення сторінок є *буферизація сторінок.* При цьому в системі підтримуються два списки фреймів — модифікованих фреймів L_m і вільних фреймів L_f. У разі заміщення сторінку не вилучають із пам'яті, а її фрейм вносять в один із цих списків: у список L_m, якщо для неї біт $M = 1$, і у список L_f, якщо $M = 0$. Насправді сторінку фізично не вилучають із її фрейму: відповідний їй запис просто вилучають із таблиці сторінок, а інформацію з її фрейму поміщають у кінець відповідного списку.

Список L_f містить фрейми, які можна використати для розміщення сторінок після сторінкового переривання. У разі необхідності такого розміщення сторінку завантажують у перший фрейм списку, колишній вміст цього фрейму затирають, а сам він вилучається із L_f (тільки в цьому разі відбувається заміщення сторінки у звичайному розумінні).

Список L_m містить фрейми із модифікованими сторінками, які мають бути записані на диск перед розміщенням у відповідних фреймах нових даних. Таке записування може відбуватися не по одній сторінці, а в пакетному режимі по кілька сторінок за раз, при цьому заощаджується час на дискові операції. Після збереження сторінки на диску відповідний фрейм переносять у список L_f.

Найважливішою властивістю цього алгоритму є те, що списки L_f і L_m працюють як кеші сторінок. Справді, сторінки, що відповідають фреймам із цих списків, продовжують перебувати в пам'яті, а отже, якщо знову виникає необхідність використати таку сторінку, усе, що потрібно зробити, — це повернути відповідний елемент назад у таблицю сторінок, оскільки всі дані цієї сторінки продовжують перебувати у відповідному фреймі.

Під час сторінкового переривання спочатку перевіряють, чи є відповідний фрейм у списку L_f або L_m, якщо є — його використовують негайно і звертання до диска не відбувається, якщо ж немає, береться перший фрейм зі списку вільних і в нього завантажують сторінку із диска.

Зазначимо, що використання буферизації сторінок дає змогу обмежитися найпростішим алгоритмом для безпосереднього вибору заміщуваної сторінки, наприклад алгоритмом FIFO. Сторінки, помилково заміщені FIFO-алгоритмом, потраплять не відразу на диск, а спочатку в один зі списків L_f або L_m, тому якщо вони незабаром знадобляться знову, то все ще міститимуться у пам'яті.

Деякі простіші реалізації об'єднують списки L_f і L_m в один. Такий список працює аналогічно до L_f, тільки у разі заміщення сторінки на початку списку при $M = 1$ вона буде спочатку записана на диск.

Буферизацію сторінок використовують майже в усіх сучасних операційних системах, приклади її реалізації в Linux і Windows XP будуть показані нижче.

9.5.7. Глобальне і локальне заміщення сторінок

Заміщення сторінок можна розділити на *глобальне і локальне*. За глобального заміщення процес може вибрати фрейм, що належить будь-якому процесові, і завантажити в нього свою сторінку. Воно може призвести до того, що одна й та сама програма виконуватиметься з різною продуктивністю в різних умовах, оскільки на заміщення сторінок відповідного процесу впливає поведінка інших процесів у системі. За локального заміщення процес може вибрати тільки свій власний фрейм (пул вільних фреймів підтримують окремо для кожного процесу). Як результат, маловикористовувані ділянки пам'яті процесу виявляються втраченими для інших процесів.

Зазвичай загальна продуктивність для глобального заміщення виявляється вищою. Глобальне заміщення також є простішим у реалізації.

У більшості сучасних ОС реалізоване глобальне заміщення сторінок або змішаний варіант, за якого більшу частину часу використовують локальне заміщення, а у разі нестачі пам'яті — глобальне.

9.5.8. Блокування сторінок у пам'яті

Дотепер ми виходили з того, що будь-яка невикористовувана сторінка може бути заміщена у будь-який момент часу. Насправді це не так. Розглянемо послідов-

ність подій, які можуть статися під час використання глобальної стратегії заміщення сторінок.

1. Процес А збирається виконати операцію введення, при цьому введені дані мають бути збережені в деякому буфері. Його розміщають у сторінці пам'яті, завантаженій в цей момент у відповідний фрейм. Пристрій введення має окремий контролер, який розуміє команду пересилання даних. Одним із параметрів цієї команди є адреса фізичної пам'яті, починаючи з якої зберігатимуться введені дані.
2. Процес А видає команду контролеру (вказавши як параметр адресу буфера) і призупиняється, чекаючи введення.
3. ОС перемикає контекст і передає керування процесу В.
4. Процес В генерує сторінкове переривання.
5. ОС виявляє, що вільних фреймів немає, тому застосовує алгоритм заміщення для пошуку сторінки, яку потрібно вивантажити на диск.
6. Внаслідок пошуку алгоритм вибирає для заміщення сторінку, де розташований буфер введення процесу А.
7. ОС заміщає сторінку, зберігаючи її дані на диску і завантажуючи у фрейм нову сторінку.
8. ОС перемикає контекст і передає керування знову процесу А.
9. Контролер починає запис даних за адресою, заданою на кроці 2.
10. Ця адреса тепер відповідає фрейму, у який завантажена сторінка процесу В, у результаті дані процесу В будуть перезаписані, і він швидше за все завершиться аварійно.

Щоб цього не сталося, сторінки, подібні до використаної для буфера введення, мають блокуватися в пам'яті (про них кажуть, що вони *пришпилені* – pinned). Звичайно таке блокування реалізоване на рівні додаткового біта в елементі таблиці сторінок. Якщо такий біт дорівнює одиниці, ця сторінка не може бути вибрана алгоритмом для заміщення сторінок, а відповідний фрейм не може стати фреймом-жертвою.

Використання блокування сторінок у пам'яті не обмежене описаною ситуацією. Приміром, код і дані ОС мають увесь час перебувати в основній пам'яті, тому всі відповідні сторінки варто заблокувати у пам'яті. Пам'ять, що не бере участі у заміщенні сторінок, називають *невивантажуваною* (nonpaged memory) на противагу *вивантажуваній* (paged memory).

Блокування сторінки у пам'яті потенційно небезпечне тим, що ця сторінка взагалі не буде розблокована і залишиться в основній пам'яті надовго. На практиці це зазвичай зводиться до обмеження блокування сторінок процесами користувача (наприклад, деякі ОС приймають «підказки» від процесів із приводу блокування сторінок, але залишають за собою право ігнорувати їх у разі нестачі пам'яті або коли процес збирається блокувати занадто багато сторінок).

Зазначимо, що іншим способом розв'язання проблеми вивантаження сторінки під час введення може бути заборона на копіювання даних контролером безпосередньо у пам'ять користувача. У цьому випадку ОС може помістити свій буфер у невивантажувану пам'ять, ввести туди дані, а потім скопіювати їх у буфер користувача.

9.5.9. Фонове заміщення сторінок

Дотепер ми припускали, що заміщення сторінок відбувається за необхідності (коли процес генерує сторінкове переривання, а ОС не знаходить вільного фрейму). Насправді із погляду продуктивності такий підхід не є оптимальним. У більшості сучасних ОС реалізується інший підхід — *фонове заміщення сторінок.*

Під час запуску ОС стартує спеціальний фоновий процес, або потік, який називають *процесом підкачування* або *сторінковим демоном* (swapper). Цей процес час від часу перевіряє, скільки вільних фреймів є в системі. Якщо їхня кількість менша за деяку допустиму межу, сторінковий демон починає заміщувати сторінки різних процесів, вивільняючи відповідні фрейми. Він продовжує займатися цим доти, поки кількість вільних фреймів не перевищить потрібної межі чи він не вивільнить деяку наперед відому кількість фреймів. Таким чином кількість вільних фреймів у системі підтримують на певному рівні, що підвищує її продуктивність.

Фонове заміщення сторінок звичайно використовують разом із буферизацією сторінок, у цьому разі сторінковий демон переносить фрейми у список L_f або зберігає на диску сторінки зі списку L_m.

9.6. Зберігання сторінок на диску

Дисковий простір, що використовують для зберігання сторінок, називають *простором підтримки* (backing store) або *простором підкачування* (swap space), а пристрій (диск), на якому він перебуває, — *пристроєм підкачування* (swap device). Цей пристрій повинен бути якомога продуктивнішим.

У більшості випадків обмін даними із простором підкачування відбувається швидше, ніж із файловою системою, оскільки дані розподіляються більшими блоками і не потрібно працювати з каталогами і файлами. Ця перевага може бути використана по-різному. Наприклад, під час запуску процесу можна заздалегідь копіювати весь його адресний простір у простір підкачування і вести весь подальший обмін даними тільки із цим простором. У результаті на диску завжди перебуватиме образ процесу. При цьому одним сторінкам на диску відповідатимуть фрейми пам'яті, іншим — ні, але кожна сторінка, завантажена у фрейм пам'яті, завжди відповідатиме сторінці на диску. Така стратегія вимагає багато місця для простору підкачування. Можна також під час першого завантаження на вимогу звертатися до файла із файлової системи та зчитувати дані з нього, а в разі заміщення сторінок зберігати їх у просторі підкачування. Тому там зберігатимуться тільки заміщені сторінки, і не кожній сторінці у пам'яті відповідатиме сторінка на диску, тобто образ процесу на диску не зберігатиметься.

9.7. Пробуксовування і керування резидентною множиною

9.7.1. Поняття пробуксовування

Пробуксовуванням (thrashing) називають стан процесу, коли через сторінкові переривання він витрачає більше часу на підкачування сторінок, аніж власне на виконання. У такому стані процес фактично непрацездатний.

Пробуксовування виникає тоді, коли процес часто вивантажує із пам'яті сторінки, які йому незабаром знову будуть потрібні. У результаті більшу частину часу такі процеси перебувають у призупиненому стані, очікуючи завершення операції введення-виведення для читання сторінки із диска. Отож, на додачу до пам'яті, за розміром порівнянної із диском, отримують пам'ять, порівнянну із диском і за часом доступу.

Назвемо деякі причини пробуксовування.

1. Процес не використовує пам'ять повторно (для нього не працює правило «дев'яносто до десяти»).
2. Процес використовує пам'ять повторно, але він надто великий за обсягом, тому його резидентна множина не поміщається у фізичній пам'яті.
3. Запущено надто багато процесів, тому їхня сумарна резидентна множина не поміщається у фізичній пам'яті.

У перших двох випадках ситуація майже не піддається виправленню, найкраща порада, яку тут можна дати, – це збільшити обсяг фізичної пам'яті. У третьому випадку ОС може почати такі дії: з'ясувати, скільки пам'яті необхідно кожному процесові, і змінити пріоритети планування так, щоб процеси ставилися на виконання групами, вимоги яких до пам'яті можуть бути задоволені; заборонити або обмежити запуск нових процесів.

9.7.2. Локальність посилань

Можна створити таку програму, яка постійно звертатиметься до різних сторінок, розкиданих великим адресним простором, генеруючи багато сторінкових переривань. Насправді реальні застосування працюють не так: вони зберігають *локальність посилань* (locality of reference), коли на різних етапах виконання процес посилається тільки на деяку невелику підмножину своїх сторінок, що є одним із наслідків відомого правила «дев'яносто до десяти».

Набір сторінок, які активно використовуються разом, називають *локальністю* (locality). Кажуть, що процес під час свого виконання переміщується від локальності до локальності. Так, у разі виклику функція визначає нову локальність, де будуть посилання на інструкції, локальні змінні та підмножину глобальних змінних. Після виходу із функції її локальність більше не використовують (хоча можна це зробити в майбутньому, якщо функція буде викликана знову). Отже, локальності визначає структура програми та її даних.

Якщо виділено достатньо пам'яті для всіх сторінок поточної локальності, сторінкові переривання до переходу до наступної локальності не генеруватимуться. Якщо пам'яті недостатньо, система перебуватиме у стані пробуксовування.

Завданням керування резидентною множиною є така його динамічна корекція, щоб у будь-який момент часу ця множина давала змогу розміщувати всі сторінки поточної локальності.

9.7.3. Поняття робочого набору. Модель робочого набору

Для того щоб коректно керувати резидентною множиною процесу, необхідно знати той набір сторінок, який йому знадобиться під час виконання. Звичайно, у повному обсязі таке знання реалізоване бути не може, бо це вимагає вміння вгадувати

майбутнє. Спроби оцінити потрібний процесу набір сторінок, ґрунтуючись на особливостях використання сторінок у недалекому минулому, привели до концепції *робочого набору* (working set).

Найважливішою характеристикою цього підходу є інтервал часу довжиною T, який відлічують від поточного моменту часу назад. Якщо T вимірюють у секундах, то цей інтервал визначає останні T секунд виконання процесу і задає вікно робочого набору. Ширину вікна завжди залишають постійною, а межі його зсуваються із часом; верхньою межею завжди є поточний момент часу, а нижньою — поточний момент мінус T.

Під робочим набором процесу розуміють усі сторінки, до яких він звертався у вікні робочого набору. У кожний новий момент часу робочий набір буде різним, оскільки вікно робочого процесу переміщується, сторінки, до яких не було доступу за час T, із набору вилучають, щойно використані сторінки додають у набір (рис. 9.8).

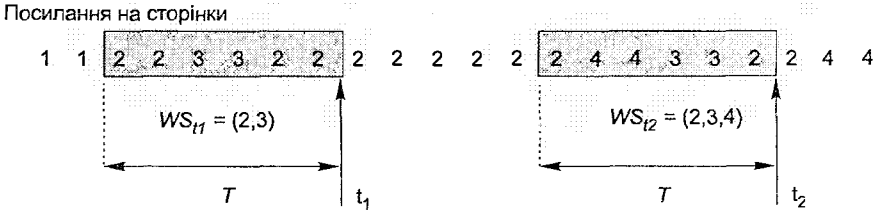

Рис. 9.8. Робочий набір процесу

Розглянемо, як можна використати концепцію робочого набору.

1. Виділяючи пам'ять, можна дати кожному процесові стільки сторінок, щоб у них помістився його робочий набір.
2. У разі заміщення сторінок можна вибирати для заміщення переважно ті сторінки, що не належать до робочого набору.
3. Під час планування процесів можна не ставити процес на виконання доти, поки всі сторінки його робочого набору не опиняться у пам'яті.

Концепцію робочого набору часто використовують у поєднанні із локальним заміщенням і буферизацією сторінок. Наприклад, можна постійно коригувати локальний кеш сторінок процесів для того щоб він приблизно відповідав за розміром його робочому набору. Цим підвищують гнучкість локального заміщення (є можливість дозволити процесам використати через певний час вільні фрейми інших процесів, зберігши при цьому ізольованість одних процесів від інших).

Головним недоліком моделі робочого процесу є складність реалізації. Вона пов'язана насамперед із тим, що вікно робочого процесу постійно переміщується, нові сторінки постійно додають у нього, старі — вилучають. Іншим недоліком є складність вибору величини T (звичайно її підбирають емпірично) і складність організації робочого набору при спільному використанні сторінками різних процесів.

Наведемо один із варіантів реалізації моделювання робочого набору. Із кожним фреймом пов'язують спеціальну характеристику — *час простою* (idle time). За перериванням від таймера перевіряють, чи встановлено біт R, якщо так, час простою і біт R обнулюють (сторінку залишають у робочому наборі), якщо ні, проміжок часу між скануваннями сторінок додають до часу простою. Коли час

простою перевищує T, сторінку виключають із робочого набору. У реальних системах значення T перебуває в межах хвилини.

Зазвичай найбільше сторінкових переривань відбувається під час завантаження процесу у пам'ять, тому багато систем намагаються відстежувати робочі набори різних процесів, аби забезпечити якнайбільше сторінок робочого набору процесу у пам'яті до того моменту, коли процес потрібно завантажити. Це ще один варіант реалізації попереднього завантаження сторінок.

9.7.4. Практичні аспекти боротьби з пробуксовуванням

Метод робочого набору відіграє важливу роль у дослідженнях поведінки процесів у системі. Однак у сучасних ОС його практичне застосування незначне через складність реалізації, а також внаслідок того, що сьогодні пробуксовування в системах трапляється все рідше і програмні методи боротьби з ним втрачають своє значення.

У більшості випадків зниження ймовірності пробуксовування пов'язане із прогресом в апаратному забезпеченні.

- У комп'ютерах зараз встановлюють більше основної пам'яті, тому потреба у підкачуванні знижується, а сторінкові переривання трапляються рідше.
- Оскільки процесори стають швидшими, процеси виконуються із більшою швидкістю, раніше вивільняючи зайняту пам'ять.
- Хоча в системі завжди є процеси, які можуть використати всю доступну пам'ять, кількість процесів, яким достатньо виділеної пам'яті, у середньому збільшується.

Фактично, найдієвіший спосіб боротьби із пробуксовуванням полягає у придбанні та встановленні додаткової основної пам'яті.

9.8. Реалізація віртуальної пам'яті в Linux

У цьому розділі розглянемо особливості керування пам'яттю Linux. Виклад торкатиметься архітектури IA-32 і версії ядра 2.4.20 [74].

9.8.1. Керування адресним простором процесу

Структура адресного простору процесу

Адресний простір процесу складається з усіх лінійних адрес, які йому дозволено використовувати. Ядро може динамічно змінювати адресний простір процесу шляхом додавання або вилучення інтервалів лінійних адрес.

Інтервали лінійних адрес зображуються спеціальними структурами даних — *регіонами пам'яті* (memory regions). Кожний регіон описують *дескриптором регіону* (`vm_area_struct`), що містить його початкову лінійну адресу, першу адресу після його кінцевої, прапорці прав доступу (читання, запис, виконання, заборона вивантаження на диск тощо). Розмір регіону має бути кратним 4 Кбайт, щоб його дані заповнювали всі призначені фрейми пам'яті. Регіони пам'яті процесу ніколи не перекриваються, ядро намагається з'єднувати сусідні регіони у більший регіон.

Усю інформацію про адресний простір процесу описують спеціальною структурою даних — *дескриптором пам'яті* (memory descriptor, `mm_struct`). Покажчик на

дескриптор пам'яті процесу зберігають у керуючому блоці процесу (task_struct). У цьому дескрипторі зберігають таку інформацію, як кількість регіонів пам'яті, покажчик на глобальний каталог сторінок, адреси різних ділянок пам'яті (коду, даних, динамічної ділянки, стека).

Крім того, у дескрипторі пам'яті процесу є покажчики на дві структури даних, призначених для забезпечення доступу до регіонів його пам'яті. Перша з них – однозв'язний список усіх регіонів процесу, який використовують для прискорення сканування всього адресного простору, друга – спеціальне бінарне дерево пошуку (що теж об'єднує всі регіони процесу), використовуване для прискорення пошуку конкретної адреси пам'яті.

Для виділення інтервалу вдаються до такого.

1. Відшукують вільний інтервал потрібної довжини (із використанням бінарного дерева).
2. Визначають регіон, що передує цьому інтервалу, і регіон, що йде за ним (коли при цьому отримують регіон, який вже використовує цей інтервал, усе починають спочатку).
3. Роблять спробу об'єднати попередній і наступний регіони із цим інтервалом. Якщо це не вдається, у пам'яті розміщують дескриптор нового регіону. Його ініціалізують адресами початку і кінця інтервалу і додають у список і дерево регіонів процесу.
4. Повертають початкову лінійну адресу нового або об'єднаного регіону.

У разі вилучення інтервалу лінійних адрес із адресного простору процесу важливо враховувати, що такий інтервал звичайно не збігається із регіоном пам'яті, він може бути частиною регіону або охоплювати кілька регіонів. Виконують такі дії.

1. Сканують список усіх регіонів пам'яті, які належать процесу, та вилучають із цього списку всі регіони, які перетинаються із заданим інтервалом.
2. Вилучені регіони скорочують або розбивають на два (залежно від характеру перетинання регіону з інтервалом).
3. Вивільняють фізичну пам'ять і змінюють таблиці сторінок процесу для вилученого інтервалу; скорочені регіони повертають назад у список і дерево регіонів процесу.

Робота з адресним простором процесу з режиму користувача

Із використанням описаних алгоритмів виділення і вилучення інтервалів лінійних адрес реалізовано засоби роботи з адресним простором процесу з режиму користувача. До цих засобів належать системний виклик `mmap()`, описаний у розділі 11; його головним призначенням є реалізація відображуваної пам'яті, а також можливість організувати виділення регіонів пам'яті заданого розміру; функції та системні виклики керування динамічною пам'яттю (`malloc()` та інші), описані в розділі 10.

Обробка сторінкових переривань

У разі спроби доступу до логічної адреси пам'яті, якій у конкретний момент не відповідає фізична адреса, виникає сторінкове переривання. Це може бути результатом помилки програміста, частиною механізму завантаження сторінок на вимогу або технології копіювання під час записування.

Коли переривання відбулося в режимі ядра, поточний процес негайно завершують (причиною переривання в цьому разі може бути передавання невірного параметра в системний виклик або помилка в коді ядра).

Коли переривання відбулося в режимі користувача, визначають, якому із регіонів пам'яті процесу відповідає лінійна адреса, що спричинила це переривання. За відсутності такого регіону процес завершують (відбулося звертання за невірною адресою). Крім того, процес завершують, якщо з'ясовується, що переривання викликане спробою записування в регіон, відкритий для читання. Завершення процесу здійснюють відсиланням йому сигналу SIGSEGV.

У разі незавершення процесу після всіх цих перевірок вважають, що він має право отримати нову сторінку. Для цього таблицю сторінок процесу перевіряють на наявність у ній сторінки, що відповідає адресі, яка викликала переривання.

- За відсутності такої сторінки, якщо вона не завантажена в жодний фрейм, ядро створює новий фрейм і завантажує в нього сторінку — так реалізують завантаження сторінок на вимогу. Якщо процес не звертався до цієї сторінки жодного разу, створюють нову, заповнену нулями, тобто необхідну сторінку завантажують із диска.
- Коли така сторінка наявна, але позначена «тільки для читання», ядро створює новий фрейм і копіює в нього її дані — так реалізують технологію копіювання під час записування.

Реалізація завантаження сторінок на вимогу в Linux пов'язана з тим, що запити процесів на розподіл динамічної пам'яті (на відміну від запитів ядра) вважають не терміновими. Наприклад, під час завантаження процесу у пам'ять імовірність того, що йому негайно знадобляться всі сторінки із програмним кодом, мала. Беручи це до уваги, ядро намагається не завантажувати сторінки у пам'ять доти, поки вони не знадобляться.

Кількість сторінкових переривань за час виконання поточного процесу можна отримати за допомогою системного виклику getrusage():

```
#include <sys/resource.h>
struct rusage usage;
getrusage(RUSAGE_SELF, &usage);  // інформація для поточного процесу
printf ("Усього сторінкових переривань: %d\n", usage.ru_majflt);
```

Керування пам'яттю під час створення і завершення процесів і потоків

У розділі 3 висвітлювалися відмінності між створенням процесів (fork()) і потоків (clone()) у Linux. Однією з найважливіших є відмінність у реалізації розподілу пам'яті.

Як відомо, системний виклик clone() приймає набір прапорців, що визначають ступінь розподілу ресурсів між потоками. Серед цих прапорців за керування пам'яттю відповідає прапорець CLONE_VM. Якщо його задано, під час створення нового потоку поле mm його керуючого блоку вказуватиме на той самий дескриптор пам'яті, що має потік (або процес), який його створив. Таким чином буде реалізовано загальний адресний простір для потоків одного процесу — усі вони використовуватимуть той самий дескриптор пам'яті.

Системний виклик fork() створює новий адресний простір процесу (при цьому жодну пам'ять не виділяють доти, поки процес не звернеться до адреси пам'яті). Спочатку створюють та ініціалізують новий дескриптор пам'яті, покажчик на нього зберігають у полі mm керуючого блоку процесу. Після цього усі дескриптори регіонів пам'яті предка додають у список регіонів пам'яті нащадка; крім того, копіюють всі таблиці сторінок предка, сторінки позначають «тільки для читання» – для предка і для нащадка (на основі цього працюватиме копіювання під час записування).

Під час завершення процесу ядро проходить за списком усі його регіони пам'яті та вивільняє всі сторінки, пов'язані з кожним регіоном. Після цього вилучають усі відповідні елементи таблиці сторінок процесу.

9.8.2. Організація заміщення сторінок

У цьому розділі розглянемо організацію заміщення сторінок у пам'яті. Відразу ж зазначимо, що в Linux реалізоване фонове заміщення сторінок, за яке відповідає потік ядра kswapd. Цей потік у колишніх версіях ядра запускався за перериванням від таймера кілька разів за секунду, у ядрі версії 2.6 необхідність його запуску визначає наявність вільних сторінок пам'яті у системі.

Списки сторінок

Організація заміщення сторінок у Linux грунтується на їх буферизації. Організовують два списки сторінок: список активних (active_list) – містить сторінки, які використовують процеси і визначає робочий набір процесів; список неактивних (inactive_list) – містить сторінки, які не так важливі для процесів (не використовуються в цей момент часу). Модифікована сторінка перебуває в списку неактивних якийсь час, перш ніж її збережуть на диску.

Нові сторінки додають у початок списку неактивних сторінок. За нестачі пам'яті частину сторінок переміщують з кінця списку активних сторінок у початок списку неактивних, а потім починають вивільнення сторінок із кінця списку неактивних сторінок (рис. 9.9).

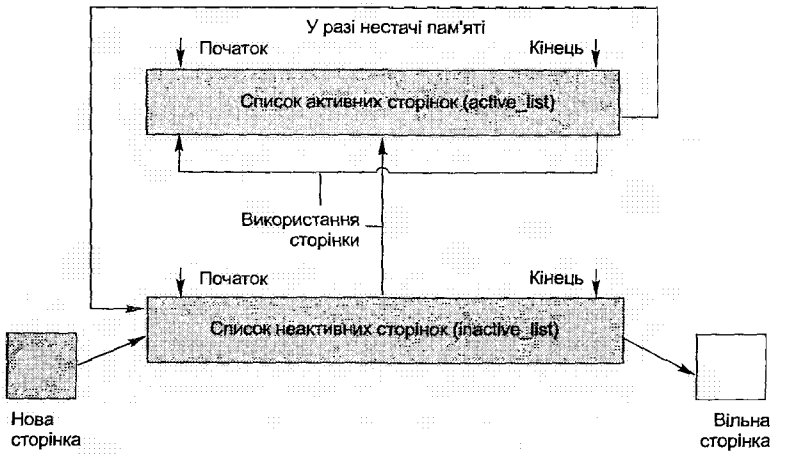

Рис. 9.9. Списки сторінок менеджера віртуальної пам'яті Linux

Визначення робочого набору

Для визначення робочого набору процесу застосовують прапорець використання сторінки (PG_referenced). Нові сторінки створюють із PG_referenced = 0. Коли процесор зчитує сторінку вперше, прапорець покладають рівним одиниці. Якщо потрібно задати PG_referenced рівним одиниці для сторінки, що перебуває у списку неактивних сторінок, його обнулюють, і сторінку переміщують у список активних (так формують робочий набір процесу). З іншого боку, неактивну сторінку можна використати для розміщення нових даних незалежно від значення цього прапорця. Якщо ж сторінка, для якої PG_referenced = 1, має бути перемішена у список неактивних сторінок, система дає змогу це зробити тільки з другої спроби обходу (сторінці дають другий шанс для її використання).

Обробка відображених, змінених і заблокованих сторінок

Під час обходу списку неактивних сторінок для вивільнення пам'яті може виникнути ситуація, коли цього відразу зробити не можна.

- Сторінка може бути відображена в адресний простір процесу або кількох процесів. Щоб вивільнити сторінку, відображення спочатку потрібно вилучити для кожного такого процесу.
- Сторінка може бути заблокована у пам'яті (наприклад, у ній виділено буфер, який використовують для введення даних із пристрою). Під час обходу таку сторінку пропускають і роблять спробу знайти іншу сторінку для вивільнення. Ця сторінка може бути знову перевірена у такому проході.
- Сторінка була модифікована, тому її спочатку треба записати на диск. Як тільки не відображену в адресний простір процесу модифіковану сторінку переміщують із початку в кінець списку неактивних сторінок, система скидає її вміст на диск.

Блокування сторінок у пам'яті

Для організації блокування сторінок у пам'яті в Linux використовують системні виклики mlock() і munlock(), прототипи яких визначені в заголовному файлі sys/mman.h. Системний виклик mlock() блокує у пам'яті набір сторінок, munlock() знімає це блокування. Обидва ці виклики приймають два параметри: адресу, з якої починається блокований набір сторінок, і розмір пам'яті, яку потрібно заблокувати або розблокувати.

Наведемо приклад блокування однієї сторінки із використанням цих викликів.

```
#include <sys/mman.h>
#include <stdlib.h>   // для malloc()
size_t pagesize = getpagesize();
// виділення пам'яті під сторінку
char* page = (char *)malloc(pagesize);
// сторінка заблокована у пам'яті й не може бути вивантажена на диск
mlock (page, pagesize);
// ...
// розблокування сторінки
munlock(page, pagesize);
```

Системний виклик mlockall() блокує у пам'яті всі сторінки поточного процесу. Користуватися ним потрібно з обережністю, оскільки заблоковану пам'ять не

можна використовувати під час розподілу пам'яті доти, поки вона не буде розблокована (або поки процес не завершиться).

9.9. Реалізація віртуальної пам'яті в Windows XP

У цьому розділі розглянемо особливості керування пам'яттю Windows XP.

9.9.1. Віртуальний адресний простір процесу

Сторінки і простір підтримки

Сторінки адресного простору процесу можуть бути *вільні* (free), *зарезервовані* (reserved) і *підтверджені* (committed).

Вільні сторінки не можна використати процесом прямо, їх потрібно спочатку зарезервувати. Це можливо зробити будь-яким процесом системи. Після цього інші процеси резервувати ту саму сторінку не можуть.

У свою чергу, процес, що зарезервував сторінку, не може цю сторінку використати до її підтвердження, тому що зв'язок із конкретними даними або програмами для неї не визначений.

Підтверджені сторінки безпосередньо пов'язані із простором підтримки на диску. Такі сторінки можуть бути двох типів.

- Дані для сторінок першого типу перебувають у звичайних файлах на диску. До таких сторінок належать сторінки коду (їм відповідають виконувані файли) і сторінки, що відповідають файлам даних, відображеним у пам'ять (див. розділ 11). Для таких сторінок простором підтримки будуть відповідні файли. Один і той самий файл може підтримувати блоки адресного простору різних процесів.
- Сторінки другого типу не пов'язані прямо із файлами на диску. Це може бути, наприклад, сторінка, що містить глобальні змінні програми. Для таких сторінок простором підтримки є спеціальний файл підкачування (swap file). У ньому не резервують простір доти, поки не з'явиться необхідність вивантаження сторінок на диск. Сторінки, зарезервовані у файлі підкачування, називають ще тіньовими сторінками (shadow pages).

На рис. 9.10 показано, як різні частини адресного простору можуть бути пов'язані з різними типами простору підтримки.

Поняття регіону пам'яті. Резервування і підтвердження регіонів

Коли для процесу виділяють адресний простір, більшу його частину становлять вільні сторінки, які не можуть бути використані негайно. Для того щоб використати блоки цього простору, процес спочатку має зарезервувати в ньому відповідні *регіони пам'яті*.

Регіон пам'яті відображає неперервний блок логічного адресного простору процесу, який може використати застосування. Регіони характеризуються початковою адресою і довжиною. Початкова адреса регіону повинна бути кратною 64 Кбайт, а його розмір — кратним розміру сторінки (4 Кбайт).

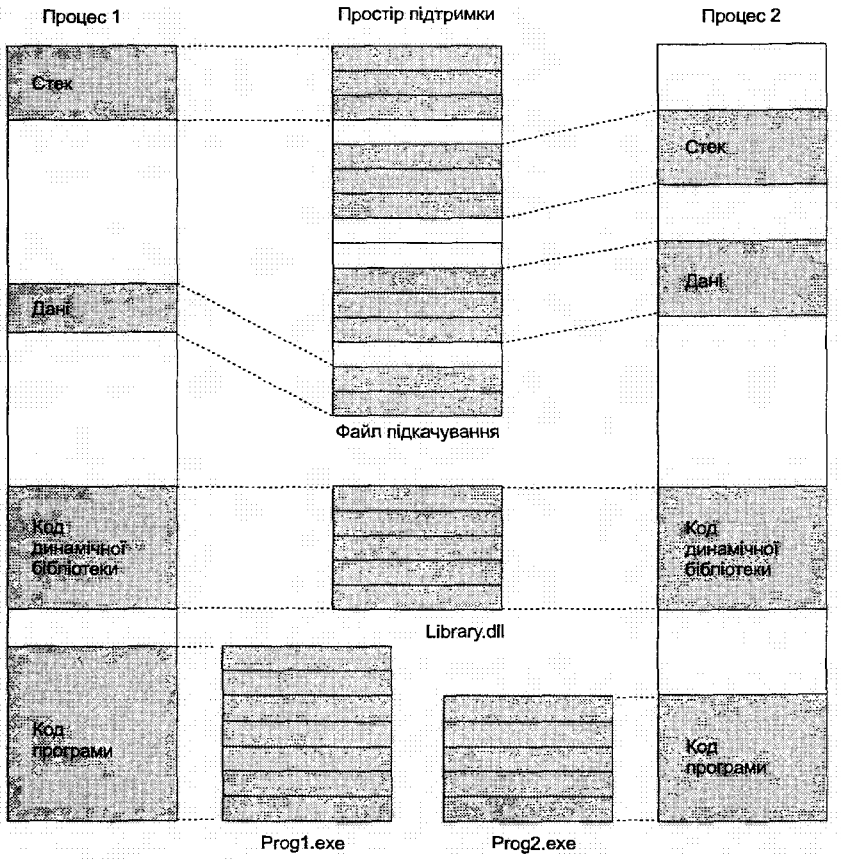

Рис. 9.10. Процеси і простір підтримки у Windows XP [44]

Для резервування регіону пам'яті використовують функцію `VirtualAlloc()` (одним із її параметрів має бути прапорець `MEM_RESERVE`). Після того як регіон був зарезервований, інші запити виділення пам'яті не можуть його повторно резервувати. Ось приклад виклику `VirtualAlloc()` для резервування регіону розміру `size`:

```
PVOID addr = VirtualAlloc(NULL, size, MEM_RESERVE, PAGE_READWRITE);
```

Першим параметром `VirtualAlloc()` є адреса пам'яті, за якою роблять виділення, якщо вона дорівнює `NULL`, пам'ять виділяють у довільній ділянці адресного простору процесу. Останнім параметром є режим резервування пам'яті, серед можливих значень цього параметра можна виділити `PAGE_READONLY` — резервування тільки для читання; `PAGE_READWRITE` — резервування для читання і записування.

Результатом виконання `VirtualAlloc()` буде адреса виділеного регіону пам'яті.

Однак і після резервування регіону будь-яка спроба доступу до відповідної пам'яті спричинятиме помилку. Щоб такою пам'яттю можна було користуватися, резервування регіону має бути підтверджене (commit). Підтвердження зводиться до виділення місця у файлі підкачування сторінок (для сторінок регіону створюють

відповідні тіньові сторінки). Для підтвердження резервування регіону теж використовують функцію VirtualAlloc(), але їй потрібно передати прапорець MEM_COMMIT:

```
VirtualAlloc(addr, size, MEM_COMMIT, PAGE_READWRITE);
```

Звичайною стратегією для застосування є резервування максимально великого регіону пам'яті, що може знадобитися для його роботи, а потім підтвердження цього резервування для невеликих блоків всередині регіону в разі необхідності. Це підвищує продуктивність, тому що операція резервування не потребує доступу до диска і виконується швидше, ніж операція підтвердження.

Можна зарезервувати і підтвердити регіон пам'яті протягом одного виклику функції

```
VirtualAlloc(addr, size, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

Після того, як використання регіону завершене, його резервування потрібно скасувати. Для цього використовують функцію

```
VirtualFree(addr, 0, MEM_RELEASE);
```

Зазначимо, що цій функції передають не розмір пам'яті, яка звільняється, а нуль (ОС сама визначає розмір регіону).

Наведемо приклад використання резервування і підтвердження пам'яті. Припустимо, що необхідно працювати із масивом у пам'яті, але в конкретний момент часу буде потрібний тільки один із його елементів. У цьому разі доцільно зарезервувати пам'ять для всього масиву, після чого підтверджувати пам'ять для окремих елементів.

```
// резервування масиву зі 100 елементів
int *array = (int *) VirtualAlloc(NULL,
        100 * sizeof(int), MEM_RESERVE, PAGE_READWRITE);
// підтвердження елемента масиву з індексом 10
VirtualAlloc(&array[10], sizeof(int), MEM_COMMIT, PAGE_READWRITE);
// робота із пам'яттю
array[10] = 200;
printf("дані у пам'яті: %d\n", array[10]);
// вивільнення масиву
VirtualFree(array, 0, MEM_RELEASE);
```

Обробка сторінкових переривань

Назвемо причини виникнення сторінкових переривань у Windows XP.

- Звертання до сторінки, що не була підтверджена.
- Звертання до сторінки із недостатніми правами. Ці два випадки є фатальними помилками і виправленню не підлягають.
- Звертання для записування до сторінки, спільно використовуваної процесами. У цьому разі можна скористатися технологією копіювання під час записування.
- Необхідність розширення стека процесу. У цьому разі оброблювач переривання має виділити новий фрейм і заповнити його нулями.
- Звертання до сторінки, що була підтверджена, але в конкретний момент не завантажена у фізичну пам'ять. Під час обробки такої ситуації використовують локальність посилань: із диска завантажують не лише безпосередньо потрібну сторінку, але й кілька прилеглих до неї, тому наступного разу їх уже не дове-

деться заново підкачувати. Цим зменшують загальну кількість сторінкових переривань.

Поточну кількість сторінкових переривань для процесу можна отримати за допомогою функції GetProcessMemoryInfo():

```
#include <psapi.h>
PROCESS_MEMORY_COUNTERS info;
GetProcessMemoryInfo(GetCurrentProcess(), &info, sizeof (info));
printf("Усього сторінкових збоїв: %d\n", info.PageFaultCount);
```

9.9.2. Організація заміщення сторінок

Базовий принцип реалізації заміщення сторінок у Windows XP – підтримка деякої мінімальної кількості вільних сторінок у пам'яті. Для цього використовують кілька концепцій: робочі набори, буферизацію, старіння, фонове заміщення і зворотне відображення сторінок.

Керування робочим набором і фонове заміщення сторінок

Поняття робочого набору є центральним для заміщення сторінок у Windows XP. У цій ОС під робочим набором розуміють множину підтверджених сторінок процесу, завантажених в основну пам'ять. Під час звертання до таких сторінок не виникатиме сторінкових переривань. Кожний набір описують двома параметрами: його нижньою і верхньою межами. Ці межі не є фіксованими, за певних умов процес може за них виходити; крім того, вони пізніше можуть мінятися. Початкове значення меж однакове для всіх процесів (нижня межа має бути в діапазоні 20–50, верхня – 45–345 сторінок).

Поняття робочого набору було описане у розділі 9.7. Очевидно, що розуміння цієї концепції у Windows XP не повністю відповідає традиційному (не використовують параметра, що задає вікно робочого набору). Фактично робочий набір Windows XP відповідає резидентній множині процесу, як це визначено в розділі 9.2. Однак у цьому розділі використовуватимемо зазначене поняття із запропонованим застереженням.

Менеджер пам'яті постійно контролює сторінкові переривання для процесу, коригуючи його робочий набір. Якщо під час обробки сторінкового переривання виявляють, що розмір робочого набору процесу менший за мінімальне значення, до цього набору додають сторінку, якщо ж більший за максимальне значення – із набору вилучають сторінку. Оскільки всі ці дії стосуються робочого набору того процесу, що викликав сторінкове переривання, базова стратегія заміщення сторінок є локальною. Втім, локальність заміщення є відносною: у деяких ситуаціях система може коригувати робочий набір одного процесу за рахунок інших (наприклад, якщо для одного процесу бракує фізичної пам'яті, а для інших її достатньо).

Крім локального коригування робочого набору в оброблювачах сторінкових переривань, у системі також реалізовано глобальне фонове заміщення сторінок. Спеціальний потік ядра, який називають *менеджером балансової множини* (balance set manager), виконується за таймером раз за секунду і перевіряє, чи не опустилася кількість вільних сторінок у системі нижче за допустиму межу. Якщо так, потік запускає інший потік ядра – *менеджер робочих наборів* (working set manager), що забирає додаткові сторінки у процесів, коригуючи їхні робочі набори.

Під час вибору сторінок для вилучення із робочого набору застосовують модифікацію годинникового алгоритму із використанням концепції старіння сторінок. Із кожною сторінкою пов'язаний цілочисловий лічильник ступеня старіння. Усі сторінки набору обходять по черзі. Якщо біт R для сторінки дорівнює 0, лічильник збільшують на одиницю, якщо R дорівнює 1, лічильник обнуляється. Внаслідок обходу заміщуються сторінки із найбільшим значенням лічильника.

Заміщені сторінки не зберігають негайно на диску, а буферизують.

Особливості буферизації сторінок

Організація буферизації сторінок у Windows XP доволі складна, але переважно дотримуються базової схеми, описаної у розділі 9.5.6.

Система підтримує 5 списків сторінок, переходи між якими показані на рис. 9.11.

- Модифікованих (modified page list). Містить вилучені із робочих наборів сторінки, які були модифіковані й котрі потрібно зберегти на диску. За принципом використання він аналогічний списку L_m.

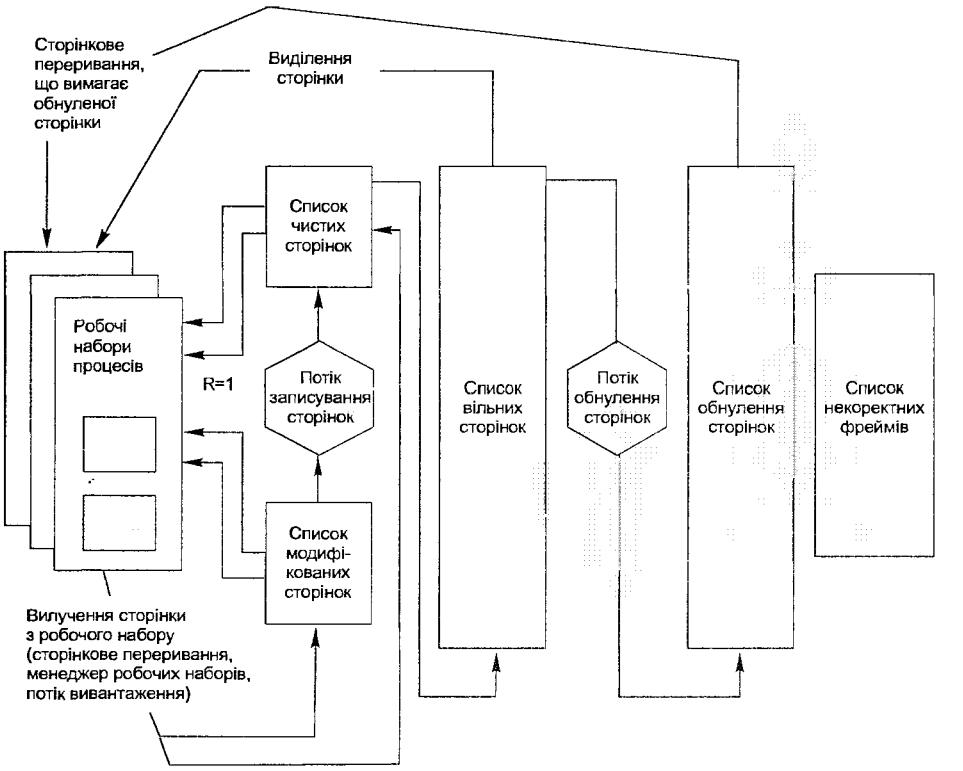

Рис. 9.11. Списки сторінок у Windows XP

- Чистих (standby page list). Містить вилучені із робочих наборів сторінки, які можна негайно використати для виділення пам'яті. Він аналогічний списку L_f. Списки чистих і модифікованих сторінок працюють як кеші сторінок (див. розділ 9.5.6).

- Вільних (free page list). У ньому перебувають сторінки, які не містять осмисленої інформації (немає сенсу повертати їх в робочі набори процесів).
- Обнулених (zero page list). Аналогічний списку вільних сторінок, але сторінки в ньому заповнені нулями.
- Некоректних фреймів фізичної пам'яті (bad page list). Фрейми в ньому відповідають ділянкам фізичної пам'яті, під час доступу до яких відбувався апаратний збій. Фрейми із цього списку ніколи не будуть використані менеджером пам'яті.

Зазначимо, що у Windows XP відсутня концепція списку активних сторінок, таким списком є робочі набори процесів.

Під час вилучення із робочого набору процесу (за сторінкового переривання або внаслідок роботи менеджера робочих потоків) сторінка потрапляє в кінець списку чистих або модифікованих сторінок (залежно від стану її біта M). Після завершення процесу в ці списки переміщують усі його сторінки.

Крім того, за переміщення сторінок між списками відповідають кілька спеціалізованих потоків ядра.

- *Потік вивантаження* (swapper), який займається вивільненням сторінок неактивних процесів. Його запускають за таймером через 4 с. Якщо цей потік знаходить процес, усі потоки якого перебували у стані очікування упродовж деякого часу (від 3 до 7 с), він переміщує всі сторінки його робочого набору у списки чистих і модифікованих сторінок.
- *Потік записування модифікованих сторінок* (modified page writer), який виконує запис модифікованих сторінок на диск. Після того, як цей потік запише сторінку на диск, її переміщують у кінець списку чистих сторінок.
- Низькопріоритетний *потік обнулiння сторінок* (zero page thread), переважно виконується, коли система не навантажена.

Коли потрібна нова сторінка, процес спочатку переглядає список вільних сторінок. Якщо він порожній, процес звертається до списку обнулених сторінок, якщо і в ньому немає жодного елемента, сторінку беруть зі списку чистих сторінок.

Попереднє завантаження сторінок

У Windows XP з'явилася підтримка попереднього завантаження сторінок. Вона ґрунтується на спостереженні за завантаженням коду програми. Воно часто сповільнюється внаслідок сторінкових переривань, які призводять до читання даних із різних файлів.

Для зменшення кількості файлів, до яких потрібно звертатися під час завантаження програмного коду, виконавча система Windows XP відслідковує сторінкові переривання упродовж 10 с під час першого запуску застосування. Після цього зібрану інформацію, зокрема, сторінки, завантажені у пам'ять, зберігають у спеціальному *файлі попереднього завантаження застосування* у підкаталозі Prefetch системного каталогу Windows XP.

Під час наступних спроб запуску застосування перевіряють, чи не був для нього уже створений файл попереднього завантаження. Якщо це так, відбувається завантаження у пам'ять збережених сторінок даних із цього файла. При цьому звертатися до файлів, з яких були завантажені ці сторінки під час першого запуску застосування, не потрібно.

Аналогічні дії відбуваються й під час завантаження всієї системи, коли створюють *файл попереднього завантаження Windows XP*. Дані з цього файла будуть зчитані у пам'ять під час наступних завантажень системи.

Блокування сторінок у пам'яті

Як і в Linux, у Windows XP можна блокувати сторінки у пам'яті з коду режиму користувача. Для цього використовують функції Win32 API `VirtualLock()` і `VirtualUnlock()`. Відповідні сторінки мають бути до цього часу підтверджені.

```
SYSTEM_INFO info;   GetSystemInfo(&info);
DWORD pagesize = info.dwPageSize;
// резервування і підтвердження сторінки
char *pageaddr = (char *)VirtualAlloc(NULL,
                pagesize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
// блокування сторінки у пам'яті
VirtualLock(pageaddr, pagesize);
// ... розблокування
VirtualUnlock(pageaddr, pagesize);
```

Оскільки блокування сторінок призводить до того, що відповідна фізична пам'ять виявляється втраченою для інших застосувань і системи, обсяг пам'яті, дозволеної для блокування, не може перевищувати нижньої межі робочого набору процесу.

Висновки

- Основною технологією взаємодії із диском в організації віртуальної пам'яті є завантаження сторінок на вимогу. При цьому сторінки не завантажують у пам'ять доти, поки до них не звернуться. Звертання до сторінки, не завантаженої у пам'ять, викликає сторінкове переривання, оброблювач такого переривання знаходить потрібну сторінку на диску і завантажує її у фізичну пам'ять. Така обробка займає багато часу, тому для досягнення прийнятної продуктивності кількість сторінкових переривань має бути якомога меншою. Невірна реалізація завантаження сторінок на вимогу може спричинити пробуксовування, коли процес витрачає більше часу на обмін із диском під час завантаження сторінок, ніж на корисну роботу.
- Коли потрібна вільна сторінка, а у фізичній пам'яті місця немає, потрібно зробити заміщення сторінки. Заміщувана сторінка буде збережена на диску, а у її фрейм завантажиться нова. Є багато алгоритмів визначення заміщуваної сторінки, найефективнішим на практиці є алгоритм LRU і його наближення, зокрема годинниковий алгоритм. На продуктивність заміщення сторінок впливають й інші фактори такі, як стратегія заміщення (глобальне або локальне) і визначення робочого набору програми (множини сторінок, які вона використовує в конкретний момент). На практиці часто використовують буферизацію сторінок, коли заміщені сторінки не записують відразу на диск, а зберігають у пам'яті у спеціальних списках, що відіграють роль кеша сторінок.

Контрольні запитання та завдання

1. У чому принципова відмінність обробки сторінкового переривання від обробки системного виклику? У якому випадку необхідно зберігати більший обсяг інформації?
2. У системі використовують дворівневі таблиці сторінок з асоціативною пам'яттю і підкачуванням на вимогу. Розрахуйте ефективний час доступу до пам'яті у випадку, якщо час доступу до основної пам'яті становить 100 нс, відсоток влучення асоціативної пам'яті – 80 %, ймовірність сторінкового переривання – 0,001, час обробки сторінкового переривання – 10 мс. Час доступу до асоціативної пам'яті вважайте нульовим.
3. Які операції з асоціативною пам'яттю має виконати ОС під час заміщення сторінки?
4. Як впливає вибір алгоритму заміщення сторінок на вибір алгоритму планування процесів?
5. Відомо, що кількість сторінкових переривань обернено пропорційна кількості доступних фреймів пам'яті. Припустімо, що сторінкове переривання в середньому обробляють 10 мс. Програма виконується в деякій ділянці пам'яті і генерує 5000 сторінкових переривань. Загальний час виконання програми – 1 хвилина. Як зміниться загальний час виконання програми, якщо обсяг доступної пам'яті збільшити вдвічі? У скільки разів потрібно збільшити обсяг пам'яті, щоб на обробку сторінкових переривань затрачалося стільки ж часу, що й на виконання інструкцій програми?
6. Якщо сторінка до моменту заміщення була змінена, її потрібно записати на диск. Як впливає розмір сторінки на кількість операцій записування змінених сторінок?
7. У деяких ОС під час роботи зі сторінками коду пристрою підкачування не використовують. Сторінки зчитують безпосередньо з файла програми, що виконується. У чому полягає недолік цього методу? Як можна його усунути?
8. Операція читання з диска на рівні контролера потребує задання адреси буфера фізичної пам'яті. Системний виклик зчитування з диска у деякій ОС приймає як параметр логічну адресу буфера, перевіряє її на коректність, перетворює у фізичну адресу і передає контролеру. У чому недоліки такої схеми і як можна її поліпшити?
9. Визначте кількість сторінкових переривань і кінцевий стан пам'яті для рядка посилань: 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6 у системі з чотирма вільними фреймами пам'яті у разі використання наступних алгоритмів:
   а) FIFO;
   б) оптимального;
   в) LRU;
   г) годинникового (без додаткових бітів).
10. Як відомо, для сторінкової організації пам'яті апаратно підтримують захист пам'яті на рівні окремих сторінок. Як програмно реалізувати задання різних

прав для ділянок пам'яті в межах однієї й тієї ж сторінки (наприклад, для першої половини сторінки – доступ для читання, для другої – для записування)?

11. Чи використовує принцип локальності доступу:
    а) LRU-алгоритм заміщення сторінок;
    б) FIFO-алгоритм заміщення сторінок?

    Поясніть Ваші відповіді.
12. Як впливає розмір сторінки на загальний розмір (у байтах) робочого набору процесу і на кількість сторінок у робочому наборі?

Розділ 10

Динамічний розподіл пам'яті

- Принципи функціонування розподілювачів пам'яті
- Організація списків вільних блоків
- Системи двійників
- Посилання та збирання сміття
- Динамічний розподіл пам'яті в Linux та Windows

Динамічний розподіл пам'яті — це розподіл за запитами процесів користувача або ядра під час їхнього виконання. Розрізняють динамічне виділення та динамічне вивільнення пам'яті. Прикладом бібліотечної та мовної підтримки засобів динамічного розподілу пам'яті є функції `malloc()`, `free()` і `realloc()` бібліотеки мови С, оператори `new` і `delete` в C++. Усі ці засоби спираються на підтримку з боку операційної системи.

Зазначимо, що алгоритми цього розділу передбачають наявність суцільного лінійного адресного простору, абстрагуючись від того, як цей простір став доступний процесу, тобто від реалізації віртуальної пам'яті. Ці алгоритми є засобами розподілу пам'яті вищого рівня порівняно із засобами реалізації віртуальної пам'яті.

З іншого боку, деякі з цих алгоритмів можна використати для реалізації розподілу не тільки логічної, але й неперервної фізичної пам'яті, зокрема й для потреб ядра системи. У цьому розділі йтиметься саме про такі алгоритми на прикладах динамічного розподілу пам'яті ядром Linux і Windows XP.

10.1. Динамічна ділянка пам'яті процесу

Динамічна ділянка пам'яті процесу відображає спеціальну частину його адресного простору, у якій відбувається розподіл пам'яті за запитами з його коду (подібно до виклику `malloc()` для виділення пам'яті або `free()` для її вивільнення). Такий розподіл пам'яті відбувається в режимі користувача, звичайно в коді системних бібліотек (цей розподіл відбувається всередині динамічної ділянки, і ядро в ньому участі не бере).

Такі операції у більшості випадків не ведуть до зміни розмірів динамічної ділянки, але якщо така зміна необхідна (наприклад, якщо вільного місця в динамічній

ділянці не залишилося), ОС пропонує для цього спеціальний системний виклик. В UNIX-системах його зазвичай називають `brk()`. Його пряме використання у прикладних програмах не рекомендоване, хоч і можливе.

Динамічний розподіл пам'яті відбувається на двох рівнях. Спочатку ядро резервує динамічну ділянку пам'яті процесу за системним викликом `brk()`. Потім усередині цієї ділянки процеси користувача можуть розподіляти пам'ять, викликаючи стандартні функції типу `malloc()` і `free()`. У разі потреби динамічну ділянку можна розширювати далі або скорочувати (знову-таки за допомогою виклику `brk()`).

10.2. Особливості розробки розподілювачів пам'яті

Розподілювач пам'яті (memory allocator) – частина системної бібліотеки або ядра системи, відповідальна за динамічний розподіл пам'яті.

Головним завданням такого розподілювача є відстеження використання процесом ділянок пам'яті в конкретний момент. Основними цілями роботи розподілювача є мінімізація втраченого внаслідок фрагментації простору і витрат часу на виконання операцій (ці характеристики є основними критеріями порівняння алгоритмів розподілу пам'яті). При цьому розподілювач має враховувати принцип локальності (блоки пам'яті, які використовуються разом, виділяються поруч один з одним).

Сучасні розподілювачі пам'яті не можуть задовольнити всі ці вимоги і є певною мірою компромісними рішеннями [61, 109]. Доведено, що для будь-якого алгоритму розподілу пам'яті A можна знайти алгоритм B, що за відповідних умов буде кращим за продуктивністю, ніж A. На практиці, однак, можуть бути розроблені алгоритми, які використовують особливості поведінки реальних програм і працюють в основному добре.

Звичайні розподілювачі не можуть контролювати розмір і кількість використовуваних блоків пам'яті: цим займаються процеси користувача, а розподілювач просто відповідає на їхні запити. Крім того, розподілювач не може займатися *дефрагментацією пам'яті* (переміщенням її блоків для того щоб зробити вільну пам'ять неперервною), оскільки для нього недопустимо змінювати покажчики на пам'ять, виділену процесові користувача (цей процес має бути впевнений у тому, що адреси, які він використовує, не змінюватимуться без його відома). Якщо було ухвалене рішення про виділення блоку пам'яті, цей блок залишається на своєму місці доти, поки застосування не вивільнить його явно. Розподілювач фактично має справу тільки із вільною пам'яттю, основне рішення, яке він має приймати, – де виділити наступний потрібний блок.

10.2.1. Фрагментація у разі динамічного розподілу пам'яті

Динамічний розподіл пам'яті може спричиняти зовнішню та внутрішню фрагментацію. Саме ступінь фрагментації є основним критерієм оцінки розподілювачів пам'яті.

Фрагментація виникає з двох причин.

- Різні об'єкти мають різний час життя (якщо об'єкти, що перебувають у пам'яті поруч, вивільняються в різний час). Розподілювачі пам'яті можуть використати цю закономірність, виділяючи пам'ять так, щоб об'єкти, які можуть бути знищені одночасно, розташовувалися поруч.
- Різні об'єкти мають різний розмір. Якби всі запити вимагали виділення пам'яті одного розміру, зовнішньої фрагментації не було б (цього ефекту домагаються, наприклад, при використанні сторінкової організації пам'яті).

Для боротьби із внутрішньою фрагментацією більшість розподілювачів пам'яті розділяють блоки пам'яті на менші частини, виділяють одну з них і далі розглядають інші частини як вільні. Крім того, часто використовують злиття сусідніх вільних блоків, щоб задовольняти запити на блоки більшого розміру.

10.2.2. Структури даних розподілювачів пам'яті

Найпростіший підхід до динамічного розподілу пам'яті реалізований у системному виклику `brk()`. Він може бути виконаний надзвичайно ефективно, тому що для його реалізації достатньо зберегти покажчик на початок вільної динамічної ділянки пам'яті та збільшувати його після кожної операції виділення пам'яті (рис. 10.1). На жаль, реалізувати вивільнення пам'яті на основі такого виклику неможливо, оскільки воно не може бути виконане послідовно в порядку, зворотному до виділення пам'яті. У результаті після вивільнення пам'яті в ній з'являтимуться «діри» (зовнішня фрагментація).

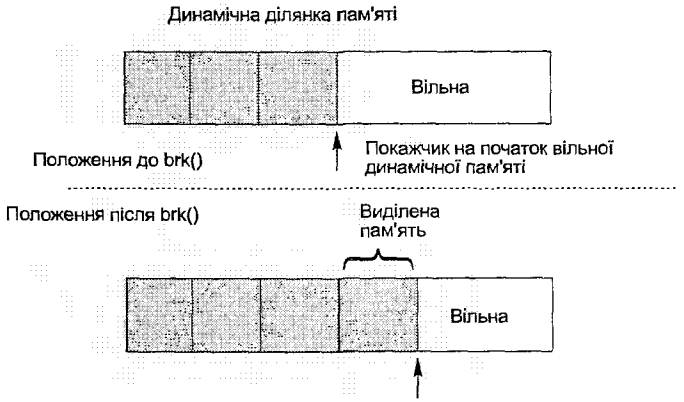

Рис. 10.1. Зміна розміру динамічної ділянки пам'яті

Покажчик, який використовується `brk()`, є найпростішим прикладом структури даних розподілювача пам'яті.

Складніші розподілювачі потребують складніших структур даних. Відстеження вільного простору у пам'яті може відбуватися за допомогою бітових карт, де кожний біт відповідає ділянці пам'яті (якщо він дорівнює одиниці, ця ділянка пам'яті зайнята, якщо нулю — вільна), або зв'язних списків зайнятих і вільних ділянок пам'яті, де кожний елемент списку містить індикатор зайнятості ділянки пам'яті та довжину цієї ділянки.

10.3. Послідовний пошук підходящого блоку

Розглянемо конкретні підходи до реалізації динамічного розподілу пам'яті. Спочатку зупинимося на алгоритмах, які зводяться до послідовного перегляду вільних блоків системи і вибору одного з них. Такі алгоритми відомі досить давно і докладно описані в літературі [41, 44, 109]. До цієї групи належать алгоритми *найкращого підходящого* (best fit), *першого підходящого* (first fit), *наступного підходящого* (next fit) і деякі інші, близькі до них. Ця група алгоритмів дістала назву алгоритми послідовного пошуку підходящого блоку (sequential fits).

10.3.1. Алгоритм найкращого підходящого

Алгоритм найкращого підходящого зводиться до виділення пам'яті із вільного блоку, розмір якого найближчий до необхідного обсягу пам'яті. Після такого виділення у пам'яті залишатимуться найменші вільні фрагменти. Структури даних вільних блоків у цьому випадку можуть об'єднуватися у список, кожний елемент якого містить розмір блоку і покажчик на наступний блок. Під час вивільнення пам'яті має сенс поєднувати суміжні вільні блоки.

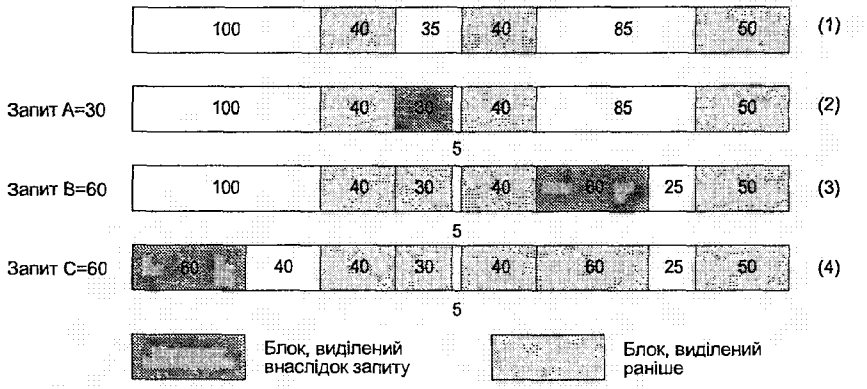

Рис. 10.2. Алгоритм найкращого підходящого

Основною особливістю розподілу пам'яті відповідно до цього алгоритму є те, що при цьому залишаються вільні блоки пам'яті малого розміру (їх називають «ошурками», sawdust), які погано розподіляються далі. Однак, практичне використання цього алгоритму свідчить, що цей недолік суттєво не впливає на його продуктивність.

10.3.2. Алгоритм першого підходящого

Алгоритм першого підходящого полягає в тому, що вибирають перший блок, що підходить за розміром (рис. 10.3). Структури даних для цього алгоритму можуть бути різними: стек (LIFO), черга (FIFO), список, відсортований за адресою. Алгоритм зводиться до сканування списку і вибору першого підхожого блоку. Якщо блок значно більший за розміром, ніж потрібно, він може бути розділений на кілька блоків.

Різні модифікації цього алгоритму на практиці виявляють себе по-різному.

Так, алгоритм першого підходящого, який використовує стек, зводиться до того, що вивільнений блок поміщають на початок списку (вершину стека). Такий підхід простий у реалізації, але може спричинити значну фрагментацію. Прикладом такої фрагментації є ситуація, коли одночасно виділяють більші блоки пам'яті на короткий час і малі блоки — на довгий. У цьому разі вивільнений великий блок буде, швидше за все, відразу використаний для виділення пам'яті відповідно до запиту на малий обсяг (і після цього не вивільниться найближчим часом).

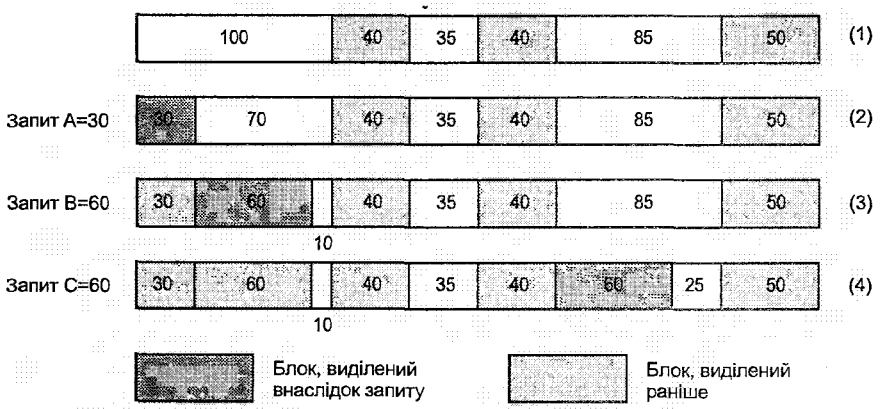

Рис. 10.3. Алгоритм першого підходящого

З іншого боку, така проблема не виникає, якщо список вільних блоків організовують за принципом FIFO (вільні блоки додають у кінець цього списку) або впорядковують за адресою. Ці модифікації алгоритму першого підходящого використовують найчастіше.

10.3.3. Порівняння алгоритмів послідовного пошуку підходящого блоку

З погляду фрагментації алгоритми найкращого підходящого та першого підходящого практично рівноцінні (фрагментацію за умов реального навантаження підтримують приблизно на рівні 20 %). Такий результат може здатися досить несподіваним, оскільки в основі цих алгоритмів лежать різні принципи. Можливе пояснення полягає в тому, що у разі використання алгоритму першого підходящого список вільних блоків згодом стає впорядкованим за розміром (блоки меншого розміру накопичуються на початку списку), тому для невеликих об'єктів (а саме такі об'єкти здебільшого виділяє розподілювач) він працює майже так само, як і алгоритм найкращого підходящого.

Алгоритм першого підхожого звичайно працює швидше, оскільки пошук найкращого підходящого вимагає перегляду всього списку вільних блоків.

Головним недоліком алгоритмів послідовного пошуку вільних блоків є їхня недостатня масштабованість у разі збільшення обсягу пам'яті. Що більше пам'яті, то довшими стають списки, внаслідок чого зростає час їхнього перегляду.

10.4. Ізольовані списки вільних блоків

Інший важливий клас алгоритмів динамічного розподілу пам'яті зводиться до організації списків вільних блоків у структуру даних, що містить масив розмірів блоків, причому кожен елемент масиву пов'язаний зі списком описувачів вільних блоків (рис. 10.4). Пошук потрібного блоку зводиться до пошуку потрібного розміру в масиві та вибору елемента із відповідного списку. Таку технологію називають технологією *ізольованих списків вільних блоків* (segregated free lists) або *ізольованою пам'яттю* (segregated storage).

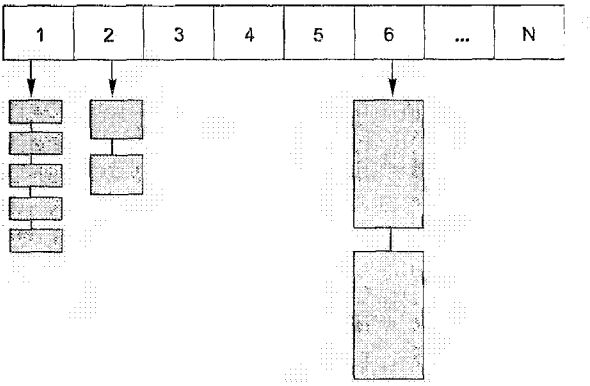

Рис. 10.4. Ізольовані списки вільних блоків

Розрізняють два базові підходи до організації такого виду списків: *проста ізольована пам'ять* (simple segregated storage) та *ізольований пошук підходящого блоку* (segregated fits).

10.4.1. Проста ізольована пам'ять

У разі використання цього підходу поділу більших блоків не відбувається, щоб задовольнити запит на виділення блоку меншого розміру. Коли під час обслуговування запиту на виділення пам'яті з'ясовують, що список вільних блоків цього розміру порожній, видають запит операційній системі на розширення динамічної ділянки пам'яті (наприклад, через системний виклик `brk()`). Після цього виділені сторінки розбивають на блоки одного розміру, які й додають у відповідний список вільних блоків. У результаті сторінка пам'яті завжди містить блоки одного розміру.

Основними перевагами такого підходу є те, що немає необхідності зберігати для кожного об'єкта інформацію про його розмір. Тепер така інформація може бути збережена для цілої сторінки об'єктів, що особливо вигідно тоді, коли розмір об'єкта малий і на сторінці таких об'єктів міститься багато.

Продуктивність цього підходу досить висока, особливо через те, що об'єкти одного розміру постійно виділяють і вивільняють упродовж малого періоду часу.

Головним недоліком простої ізольованої пам'яті є доволі відчутна зовнішня фрагментація. Не робиться жодних спроб змінювати розмір блоків (розбивати на менші або поєднувати) для того щоб задовольняти запити на блоки іншого розміру. Наприклад, якщо програма запитуватиме блоки тільки одного розміру, система із

вичерпуванням їхнього запасу розширюватиме динамічну пам'ять, незважаючи на наявність вільних блоків інших розмірів. Одним із компромісних підходів є відстеження кількості об'єктів, виділених на кожній сторінці, і вивільнення сторінки (можливо, для подальшого розміщення об'єктів іншого розміру) у разі відсутності об'єктів.

10.4.2. Ізольований пошук підходящого блока

Цей підхід також підтримує структуру даних, що складається із масиву списків вільних блоків, але організація пошуку виглядає інакше. Пошук у масиві відбувається за правилами для послідовного пошуку підходящого блоку. Якщо блок потрібного розміру не знайдено, алгоритм намагається знайти блок більшого розміру і розбити його на менші. Запит на розширення динамічної ділянки пам'яті виконують лише тоді, коли жоден блок більшого розміру знайти не вдалося.

Розрізняють три категорії підходів ізольованого пошуку підхожого блоку.

- Під час використання підходу точних списків підтримують масив списків для кожного можливого розміру блоку. Такий масив може бути досить великим, але достатньо зберігати тільки ті його елементи, для яких справді задані списки.
- Підхід точних класів розмірів із заокругленням зводиться до зберігання в масиві обмеженого набору розмірів (наприклад, ступенів числа 2) і пошуку найближчого підходящого. Цей підхід ефективніший, але більш схильний до внутрішньої фрагментації.
- Підхід класів розмірів зі списками діапазонів зводиться до того, що в кожному списку містяться описувачі вільних блоків, довжина яких потрапляє в деякий діапазон. Це означає, що блоки у списку можуть бути різного розміру (у межах діапазону). Для пошуку у списку використовують алгоритм найкращого підходящого або першого підходящого.

10.5. Системи двійників

Система двійників (buddy system) [41, 109] – підхід до динамічного розподілу пам'яті, який дає змогу рідше розбивати на частини більші блоки для виділення пам'яті під блоки меншого розміру, знижуючи цим зовнішню фрагментацію. Вона містить у собі два алгоритми: виділення та вивільнення пам'яті. Розглянемо найпростішу *бінарну* систему двійників (binary buddy system).

У разі використання цієї системи пам'ять розбивають на блоки, розмір яких є степенем числа 2: 2^K, $L \le K \le U$, де 2^L – мінімальний розмір блоку; 2^U – максимальний розмір блоку (він може бути розміром доступної пам'яті, а може бути й меншим).

Алгоритм виділення пам'яті

Опишемо принцип роботи алгоритму виділення пам'яті.

1. Коли надходить запит на виділення блоку пам'яті розміру M, відбувається пошук вільного блоку підходящого розміру (такого, що $2^{K-1} \le M \le 2^K$). Якщо блок такого розміру є, його виділяють.

2. У разі відсутності блоку такого розміру беруть блок розміру 2^{K+1}, ділять навпіл на два блоки розміру 2^K і перший із цих блоків виділяють; другий залишається вільним і стає *двійником* (buddy) першого. Робота алгоритму на цьому завершується.
3. За відсутності блоку розміром 2^{K+1} беруть найближчий вільний блок, більший за розміром від M, наприклад блок розміру 2^{K+N}. Він стає поточним блоком. Якщо немає жодного блоку, більшого за M, повертають помилку.
4. Після цього починають рекурсивний процес розподілу блоку. На кожному кроці цього процесу поточний блок ділиться навпіл, два отриманих блоки стають двійниками один одного, після цього перший із них стає поточним (і ділиться далі), а другий залишають вільним і надалі не розглядають. Для блоку розміру 2^{K+N} процес завершують через N кроків поділу отриманням двох блоків розміру 2^K. Перший із цих блоків виділяють, другий залишають вільним. Внаслідок поділу отримують N пар блоків-двійників.

Для ілюстрації цього алгоритму наведемо приклад (рис. 10.5).

1. Припустимо, що в системі є вільний блок розміром 512 Кбайт (1) і надійшов запит на виділення блоку на 100 Кбайт (блоку A).
2. Ділимо блок на два по 256 Кбайт, з них перший робимо поточним, а другий залишаємо його двійником (2). Після цього знову ділимо перший блок на два по 128 Кбайт. Це – потрібний розмір, тому виділяємо для A перший із цих блоків, а другий залишаємо його двійником (3). Тепер маємо один вільний блок на 128 Кбайт (двійник виділеного блоку) і один – на 256 Кбайт. У результаті для блоку A виділено 128 Кбайт.
3. Тепер надходить запит на виділення блоку на 30 Кбайт (блоку B). У цьому разі найближчим за розміром більшим вільним блоком буде блок на 128 Кбайт; система розділить його на два двійники по 64 Кбайт (4), а потім перший з них – на два по 32 Кбайт. Це – потрібний розмір, тому для B буде виділено перший із цих блоків, а другий залишать його двійником (5). У результаті для блоку B виділено 32 Кбайт.
4. Нарешті, надходить запит на виділення блоку на 200 Кбайт (блоку C). У цьому разі є підходящий блок на 256 Кбайт, тому його негайно виділяють (6).

Алгоритм вивільнення пам'яті

Алгоритм вивільнення пам'яті використовує утворення двійників у процесі виділення пам'яті (насправді двійниками можуть вважатися будь-які два суміжні блоки однакового розміру).

1. Заданий блок розміру 2^K вивільняють.
2. Коли цей блок має двійника і він вільний, їх об'єднують в один блок удвічі більшого розміру 2^{K+1}.
3. Якщо блок, отриманний на кроці 2, має теж двійника, їх об'єднують у блок розміру 2^{K+2}.
4. Цей процес об'єднання блоків повторюють доти, поки не буде отримано блок, для якого не знайдеться вільного двійника.

Розглянемо, як вивільнятиметься пам'ять у наведеному раніше прикладі.

1. Спочатку вивільнимо пам'ять з-під блоку B. Для нього є вільний двійник, тому їх об'єднують в один блок розміру 64 Кбайт.
2. Для цього блоку також є вільний двійник (див. (4)), і їх теж об'єднують у блок розміру 128 Кбайт (7).
3. Тепер вивільняють пам'ять з-під блоку C. Він вільних двійників не має, тому об'єднання не відбувається (8).
4. Нарешті вивільняють пам'ять з-під блоку A. Його об'єднують із двійником у блок розміру 256 Кбайт, але в того теж є вільний двійник (вивільнений з-під C), його об'єднують з тим; у результаті знову виходить один вільний блок на 512 Кбайт (9).

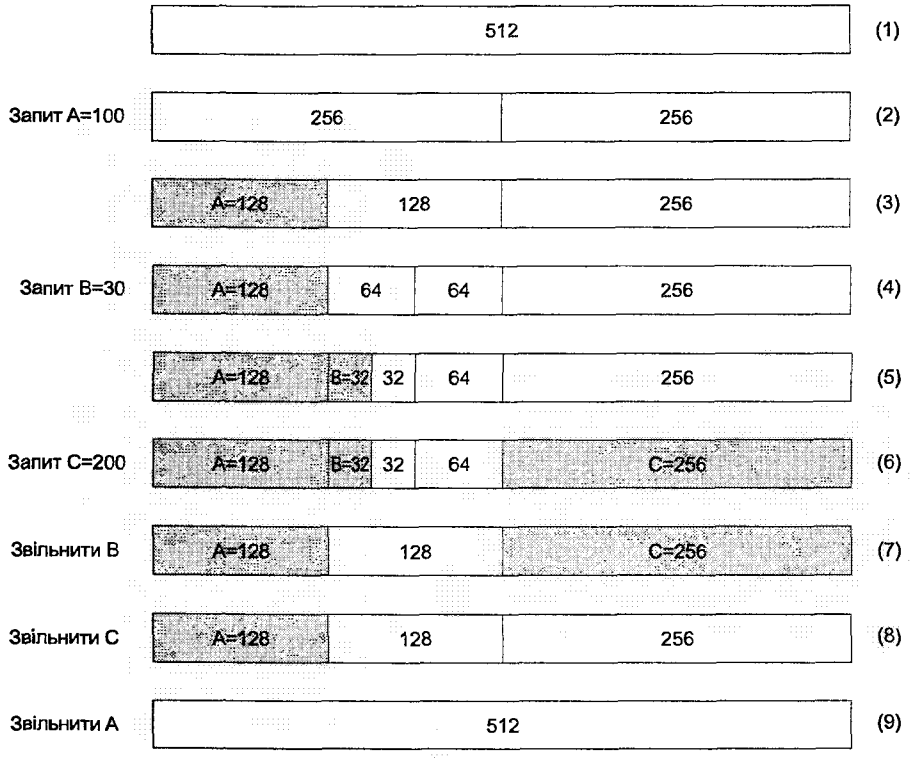

Рис. 10.5. Система двійників

Переваги і недоліки системи двійників

Цей підхід за продуктивністю випереджає інші алгоритми динамічного розподілу, він особливо ефективний для великих масивів пам'яті. Головним його недоліком є значна внутрішня фрагментація (як можна помітити, у наведеному прикладі було виділено на 86 Кбайт пам'яті більше, ніж потрібно), тому для розподілу блоків малого розміру частіше використовують інші підходи, зокрема розглянуті раніше методи послідовного пошуку або ізольованих списків вільних блоків.

10.6. Підрахунок посилань і збирання сміття

Опишемо реалізацію вивільнення пам'яті, зайнятої об'єктами. Основна проблема, що виникає при цьому, пов'язана із необхідністю відслідковувати використання кожного об'єкта у програмі. Це потрібно для того щоб визначити, коли можна вивільнити пам'ять, яку займає об'єкт. Для реалізації такого відстеження використовують два основні підходи: *підрахунок посилань* (reference counting) і *збирання сміття* (garbage collection).

10.6.1. Підрахунок посилань

Підрахунок посилань зводиться до того, що для кожного об'єкта підтримують внутрішній лічильник, збільшуючи його щоразу при заданні на нього посилань і зменшуючи при їх ліквідації. Коли значення лічильника дорівнює нулю, пам'ять з-під об'єкта може бути вивільнена.

Цей підхід добре працює для структур даних з ієрархічною організацією і менш пристосований до рекурсивних структур (можуть бути ситуації, коли кілька об'єктів мають посилання один на одного, а більше жоден об'єкт із ними не пов'язаний; за цієї ситуації такі об'єкти не можуть бути вивільнені).

10.6.2. Збирання сміття

Тут розглянемо збирання *сміття із маркуванням і очищенням* (mark and sweep garbage collection). Така технологія дає змогу знайти всю пам'ять, яку може адресувати процес, починаючи із деяких заданих покажчиків. Усю іншу пам'ять при цьому вважають недосяжною, і вона може бути вивільнена. Алгоритм такого збирання сміття виконують у два етапи.

1. На етапі маркування виділяють спеціальні адреси пам'яті, які називають кореневими адресами. Як джерела для таких адрес звичайно використовують усі глобальні та локальні змінні процесу, які є покажчиками. Після цього всі об'єкти, які містяться у пам'яті за цими кореневими адресами, спеціальним чином позначаються. Далі аналогічно позначають всю пам'ять, яка може бути адресована покажчиками, що містяться у позначених раніше об'єктах, і т. д.
2. На етапі збирання сміття деякий фоновий процес (збирач сміття) проходить всіма об'єктами і вивільняє пам'ять з-під тих із них, які не були позначені на етапі маркування.

10.7. Реалізація динамічного керування пам'яттю в Linux

У цьому розділі опишемо особливості динамічного керування пам'яттю Linux.

Спочатку зупинимося на динамічному розподілі пам'яті ядром системи [74]. Як зазначалося, деяку частину фізичної пам'яті ядро використовує для розміщення свого коду і його статичних структур даних. Цю пам'ять називають статичною пам'яттю, вона закріплена за ядром постійно, відповідні фрейми не можуть бути вивільнені.

Уся інша пам'ять є динамічною і може бути розподілена на вимогу. Тут охарактеризуємо деякі методи, які використовуються ядром для реалізації такого розподілу.

10.7.1. Розподіл фізичної пам'яті ядром

Для динамічного розподілу великих блоків фізичної пам'яті (за розміром кратних фрейму) у Linux використовують систему двійників. Використання таких блоків дає змогу знизити необхідність модифікації таблиць сторінок під час роботи застосування, а це, в свою чергу, знижує ймовірність очищення кеша трансляції.

Розподіл об'єктів ядром

Алгоритм двійників не підходить для розподілу блоків пам'яті довільного розміру, оскільки мінімальний обсяг пам'яті, який він може виділити або вивільнити, становить один фрейм (4 Кбайт), що спричиняє значну внутрішню фрагментацію.

Для розподілу пам'яті під окремі об'єкти застосовують *кусковий розподілювач* (slab allocator) [5, 61, 74]. При його розробці намагаються організувати кешування часто використовуваних об'єктів ядра в ініціалізованому стані (оскільки більша частина часу під час розміщення об'єкта витрачається на його ініціалізацію, а не на саме виділення пам'яті), а також виділення пам'яті блоками малого розміру без внутрішньої фрагментації, властивої алгоритму системи двійників.

Кеші об'єктів та їхні види

Структура даних кускового розподілювача складається зі змінної кількості кешів об'єктів, об'єднаних у двозв'язний циклічний список, який називають ланцюжком кешів. Кожний такий кеш відповідає за розподіл об'єктів конкретного типу або конкретного розміру.

Розрізняють два види кешів об'єктів.

- Спеціалізовані кеші, які створюють різні компоненти ядра системи для зберігання об'єктів конкретного типу. Для таких кешів звичайно задають *конструктор* і *деструктор*, а також унікальне ім'я, що залежить зазвичай від типу об'єкта. Під конструктором розуміють функцію, яку викликають під час ініціалізації об'єктів цього типу, під деструктором — функцію, яку викликають під час вивільнення пам'яті з-під об'єкта. Прикладом можуть бути такі кеші, як `uid_cache` (кеш ідентифікаторів користувачів), `vm_area_struct` (кеш дескрипторів регіонів) тощо. Компоненти режиму ядра (наприклад, драйвери пристроїв) можуть створювати додаткові кеші для своїх об'єктів.
- Кеші загального призначення, які використовують для зберігання блоків пам'яті довільного призначення конкретного розміру. Є кеші для блоків розміром від $2^5 = 32$ біт до $2^{17} = 13\ 1072$ біт, їх називають *size-N* (N — розмір блоку кеша у байтах, наприклад, size-128).

Кускові блоки

Пам'ять для кеша виділяють у вигляді *кускових блоків* (slabs). Кусковий блок складається із одного або кількох неперервно розташованих фреймів пам'яті, розділених на фрагменти пам'яті (chunks) одного розміру, які містять об'єкти (рис. 10.6). Розмір фрагмента пам'яті залежить від типу кеша, для якого виділяють кусковий блок (він дорівнює розміру об'єкта, екземпляри якого потрібно розподіляти за

допомогою цього кеша). Використання таких блоків для розподілу пам'яті знижує як зовнішню, так і внутрішню фрагментацію.

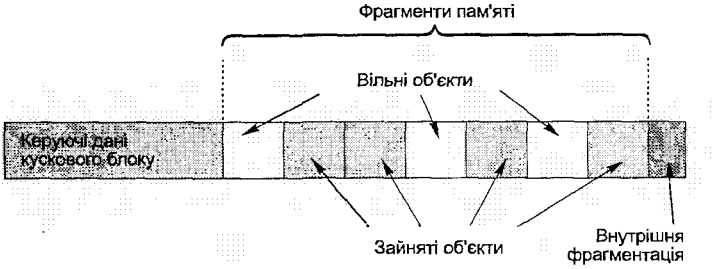

Рис. 10.6. Структура кускового блоку розподілювача об'єктів Linux

Під час створення кусковий блок негайно розділяють на певну кількість фрагментів, при цьому, якщо для відповідного кеша був заданий конструктор, його викликають для ініціалізації всіх виділених фрагментів, які перетворюються на ініціалізовані й готові до використання об'єкти. Після цього запит на виділення пам'яті під відповідний об'єкт може бути негайно задоволений, при цьому фактично ані виділення пам'яті, ані ініціалізації не відбувається (об'єкт просто позначають як виділений).

За запитом на вивільнення пам'яті об'єкт позначають як вивільнений, але з кускового блоку не вилучають і деструктор для нього не викликають; згодом його можна використати знову. Деструктор для всіх об'єктів блоку викликають тільки під час справжнього вивільнення пам'яті з-під кускового блоку або всього кеша. Зазначимо, що таке вивільнення відбувається тільки тоді, коли в цьому блоці не виділено жодного об'єкта, а система відчуває нестачу пам'яті.

Із поточним станом кешів об'єктів можна ознайомитися, виконавши команду

```
$ cat /proc/slabinfo
```

Внаслідок її виконання буде відображена таблиця, стовпчиками якої є ім'я кеша, кількість розміщених об'єктів, кількість усіх об'єктів у кускових блоках, розмір об'єкта, кількість непорожніх кускових блоків, загальна кількість кускових блоків і кількість фреймів у блоці:

```
vm_area_struct     1060   1180     64   20   20    1
size-8192             3      3   8192    3    3    2
size-32            5607   6215     32   50   55    1
```

10.7.3. Керування динамічною ділянкою пам'яті процесу

Кожний процес має доступ до спеціальної ділянки пам'яті, яку називають *динамічною ділянкою пам'яті*, або *кучею* (heap). Початкова і кінцева адреси цієї ділянки містяться в полях `start_brk` і `brk` дескриптора пам'яті процесу.

Для доступу до динамічної ділянки пам'яті використовують системний виклик `brk()`. Він намагається змінити її розмір, за параметр приймає нову кінцеву адресу цієї ділянки.

```
#include <unistd.h>
int brk(void *new_brk);
```

Зміна може зводитися до розширення або скорочення динамічної ділянки пам'яті. У разі збільшення ділянки, виконують різні перевірки (чи коректна нова адреса, чи дозволено процесу використати такий обсяг пам'яті) і після цього виділяють новий інтервал адрес, як описано у розділі 9.8.1, у разі зменшення, зайву пам'ять вивільняють. Якщо виклик `brk()` завершився коректно, поле `brk` дескриптора пам'яті відображатиме новий розмір динамічної ділянки.

Виклик `brk()` мало підходить для безпосереднього використання у прикладних програмах, оскільки з його допомогою не можна визначити поточне значення межі `brk`. На практиці як засіб керування межею динамічної ділянки використовують пакувальник цього виклику – функцію `sbrk()`.

```
#include <unistd.h>
void *sbrk(ptrdiff_t increment);
```

Вона переміщує межу динамічної ділянки на ціле число байтів `increment` і повертає покажчик на початок виділеної ділянки (фактично, старе значення межі). Якщо `increment` дорівнює нулю, `sbrk()` повертає поточне значення межі динамічної ділянки пам'яті.

```
char *oldbrk, *newbrk;
// переміщення межі динамічної ділянки
oldbrk = (char *)sbrk(10000);
// визначення нової межі динамічної ділянки
newbrk = (char *)sbrk(0);
printf("старе значення brk: %d, нове: %d\n", oldbrk, newbrk);
```

Із використанням виклику `brk()` у бібліотеці мови C для Linux реалізовані стандартні функції роботи із динамічною пам'яттю `malloc()`, `calloc()`і `realloc()`. Зазначимо, під час виклику цих функцій у режимі ядра відбувається тільки зміна `brk` на величину, кратну 4 Кбайт (у разі необхідності), подальше виділення фрагмента пам'яті потрібного розміру здійснюють у режимі користувача.

10.8. Реалізація динамічного керування пам'яттю в Windows XP

Windows XP реалізує два способи динамічного розподілу пам'яті для використання в режимі ядра: *системні пули пам'яті* (system memory pools) і *списки передісторії* (look-aside lists).

10.8.1. Системні пули пам'яті ядра

Розрізняють невивантажувані (nonpaged) і вивантажувані (paged) системні пули пам'яті. Обидва види пулів перебувають у системній ділянці пам'яті та відображаються в адресний простір будь-якого процесу.

- Невивантажувані містять діапазони адрес, які завжди відповідають фізичній пам'яті, тому доступ до них ніколи не спричиняє сторінкового переривання.
- Вивантажувані відповідають пам'яті, сторінки якої можуть бути вивантажені на диск.

Обидва види системних пулів можна використати для виділення блоків пам'яті довільного наперед невідомого розміру.

10.8.2. Списки передісторії

Списки передісторії є швидшим способом розподілу пам'яті і багато в чому схожі на кеші кускового розподілювача Linux.

Вони дають змогу виділяти пам'ять для блоків одного наперед відомого розміру. Звичайно їх спеціально створюють для виділення часто використовуваних об'єктів (наприклад, такі списки формують різні компоненти виконавчої системи для своїх структур даних). Як і для кешів кускового розподілювача, є списки загального призначення для виділення блоків заданого розміру. У разі вивільнення об'єктів вони повертаються назад у відповідний список. Якщо список довго не використовували, його автоматично скорочують.

10.8.3. Динамічні ділянки пам'яті

У Windows XP, як і в Linux, кожний процес має доступ до спеціальної ділянки пам'яті. Її також називають динамічною ділянкою пам'яті або кучею (heap). Особливістю цієї ОС є те, що таких динамічних ділянок для процесу може бути створено кілька, і всередині кожної з них розподілювач пам'яті може окремо виділяти блоки меншого розміру [31, 50].

Розподілювач пам'яті у Windows XP називають *менеджером динамічних ділянок пам'яті* (heap manager). Цей менеджер дає змогу процесам розподіляти пам'ять блоками довільного розміру, а не тільки кратними розміру сторінки, як під час керування регіонами.

Кожний процес запускають із динамічною ділянкою за замовчуванням, розмір якої становить 1 Мбайт (під час компонування програми цей розмір може бути змінений). Для роботи із цією ділянкою процес має спочатку отримати її дескриптор за допомогою виклику `GetProcessHeap()`.

```
HANDLE default_heap = GetProcessHeap();
```

Виділення блоків пам'яті

Для виділення блоків пам'яті використовують функцію `HeapAlloc()`, для вивільнення — `HeapFree()`.

```
// виділення в ділянці heap блоку пам'яті розміру size
LPVOID addr = HeapAlloc(heap, options, size);
// вивільнення в ділянці heap блоку пам'яті за адресою addr
HeapFree(heap, options, addr);
```

Першим параметром для цих функцій є дескриптор динамічної ділянки, одним із можливих значень параметра `options` для `HeapAlloc()` є `HEAP_ZERO_MEMORY`, який свідчить, що виділена пам'ять буде ініціалізована нулями.

Додаткові динамічні ділянки

Для своїх потреб процес може створювати додаткові динамічні ділянки за допомогою функції `HeapCreate()` і знищувати їх за допомогою `HeapDestroy()`.

Функція HeapCreate() приймає три цілочислові параметри і повертає дескриптор створеної ділянки.

```
HANDLE heap = HeapCreate(options, initial_size, max_size);
```

Перший параметр задає режим створення ділянки (нульове значення задає режим за замовчуванням), другий — її початковий розмір, третій — максимальний. Якщо max_size дорівнює нулю, динамічна ділянка може збільшуватися необмежено. Задання ділянок обмеженого розміру дає змогу уникнути помилок, пов'язаних із втратами пам'яті (втрата у кожному разі не зможе перевищити максимального розміру ділянки).

Розглянемо приклад використання додаткової динамічної ділянки пам'яті для розміщення масиву зі 100 елементів типу int.

```
int array_size = 100 * sizeof(int);
// розміщення динамічної ділянки
HANDLE heap = HeapCreate(0, array_size, array_size);
// розміщення масиву в цій ділянці, заповненого нулями
int *array = (int *)HeapAlloc(heap, HEAP_ZERO_MEMORY, array_size);
// робота із масивом у динамічній ділянці
array[20] = 1;
// знищення динамічної ділянки та вивільнення пам'яті
HeapDestroy(heap);
```

Зазначимо, що виклик HeapFree() у цьому прикладі не потрібний, тому що виклик HeapDestroy() призводить до коректного вивільнення всієї пам'яті, виділеної в heap. Це є однією із переваг використання окремих динамічних ділянок.

Висновки

- Динамічний розподіл пам'яті — це керування ділянкою пам'яті, якою процес може розпоряджатися на свій розсуд. Засоби керування динамічною пам'яттю працюють винятково з логічними адресами і ґрунтуються на реалізації віртуальної пам'яті. Основною проблемою динамічного розподілу пам'яті є фрагментація. Для боротьби з нею використовують різні підходи.
- Найбільш розповсюдженими технологіями динамічного розподілу пам'яті є система двійників, динамічні списки вільних блоків і послідовний пошук підходящого вільного блоку.

Контрольні запитання та завдання

1. Динамічна ділянка пам'яті процесу становить один неперервний блок розміром 256 байт. Опишіть кроки, які знадобляться для виділення послідовності блоків пам'яті розміром 7, 26, 34, 19 байт за допомогою алгоритму системи двійників. Які блоки залишаться вільними?
2. Наведіть початкову конфігурацію динамічної ділянки пам'яті і послідовність виділених блоків:
   а) якщо використання алгоритму першого підходящого призводитиме до меншої фрагментації, ніж алгоритм найкращого підходящого;

б) якщо використання алгоритму найкращого підходящого призводитиме до меншої фрагментації, ніж алгоритм першого підходящого.

3. Наведіть приклад коду, у якому неявно передбачено, що розподілювач пам'яті не буде займатися дефрагментацією динамічної ділянки пам'яті процесу.
4. Який вид фрагментації має місце:
   а) для сегментації;
   б) для сторінкової організації пам'яті;
   в) у керуванні динамічною пам'яттю процесу?
5. У якому випадку потрібно надавати більшої уваги можливій фрагментації: у випадку реалізації сторінкової організації пам'яті чи у випадку реалізації динамічного розподілу пам'яті процесу? Поясніть свою відповідь.
6. Як впливає ступінь фрагментації динамічної ділянки пам'яті процесу на кількість сторінкових переривань, що виникають під час його виконання?
7. Чим відрізняються дії, які повинні виконувати динамічні розподілювачі ядра і режиму користувача у разі нестачі пам'яті?
8. Для яких структур даних використання збирання сміття завжди приводить до коректного звільнення пам'яті, а підрахунок посилань – ні?
9. Опишіть причини, за якими системний виклик `brk()` повертає помилку. Чи можлива фрагментація пам'яті у разі його використання?
10. Виділіть спільні риси і відмінності в реалізації:
    а) керування адресним простором процесу;
    б) динамічного керування пам'яттю ядром;
    в) алгоритму заміщення сторінок;
    г) буферизації сторінок
    у Linux і Windows XP.
11. Сформуйте структуру даних для завдання 10 розділу 5 з використанням функцій керування динамічною областю пам'яті Win32 API. Для кожного екземпляра структури виділіть окрему динамічну ділянку пам'яті.

Розділ 11

Логічна організація файлових систем

- ✦ Структури файлів і файлових систем
- ✦ Організація файлової системи
- ✦ Атрибути файлів
- ✦ Операції над файлами і каталогами
- ✦ Міжпроцесова взаємодія у файловій системі

Файлові системи можна розглядати на двох рівнях: *логічному* і фізичному. Логічний визначає відображення файлової системи, призначене для прикладних програм і користувачів, фізичний – особливості розташування структур даних системи на диску й алгоритми, які використовують під час доступу до інформації.

У цьому розділі зупинимося на логічному відображенні і програмному інтерфейсі файлової системи, а в наступному – на її фізичній реалізації.

11.1. Поняття файла і файлової системи

У цьому розділі охарактеризуємо базові поняття, що лежать в основі всієї концепції файлових систем.

11.1.1. Поняття файла

Файл – це набір даних, до якого можна звертатися за іменем. Файли організовані у *файлові системи*. З погляду користувача файл є мінімальним обсягом даних файлової системи, з яким можна працювати незалежно. Наприклад, користувач не може зберегти дані на зовнішньому носії, не звернувшися при цьому до файла.

Розглянемо особливості використання файлів.

- ✦ Файли є найпоширенішим засобом зберігання інформації в енергонезалежній пам'яті. Така пам'ять надійніша, й інформація на ній може зберігатися так довго, як це необхідно. Зазначимо, що більшість збоїв у роботі ОС не руйнує інформації, що зберігається у файлах на диску. Для забезпечення збереження даних підвищеної цінності вживають додаткових заходів (гаряче резервування, резервне копіювання тощо).

- Файли забезпечують найпростіший варіант спільного використання даних різними застосуваннями. Це пов'язано з тим, що файли відокремлені від програм, які їх використовують: будь-яка програма, якій відоме ім'я файла, може отримати доступ до його вмісту. Якщо одна програма запише у файл, а інша його потім прочитає, то ці дві програми виконають обмін даними.

11.1.2. Поняття файлової системи

Файлова система – це підсистема ОС, що підтримує організований набір файлів, здебільшого у конкретній ділянці дискового простору (логічну структуру); низькорівневі структури даних, використовувані для організації цього простору у вигляді набору файлів (фізичну структуру); програмний інтерфейс файлової системи (набір системних викликів, що реалізують операції над файлами).

Файлова система надає прикладним програмам абстракцію файла. Прикладні програми не мають інформації про те, як організовані дані файла, як знаходять відповідність між ім'ям файла і його даними, як пересилають дані із диска у пам'ять тощо – усі ці операції забезпечує файлова система.

Важливо зазначити, що файлові системи можуть надавати інтерфейс доступу не тільки до диска, але й до інших пристроїв. Є навіть файлові системи, які не зберігають інформацію, а генерують її динамічно за запитом. Втім, для прикладних програм усі такі системи мають однаковий вигляд.

До головних задач файлової системи можна віднести: організацію її логічної структури та її відображення на фізичну організацію розміщення даних на диску; підтримку програмного інтерфейсу файлової системи; забезпечення стійкості проти збоїв; забезпечення розподілу файлових ресурсів за умов багатозадачності та захисту даних від несанкціонованого доступу.

11.1.3. Типи файлів

Раніше ОС підтримували файли різної спеціалізованої структури. Сьогодні є тенденція взагалі не контролювати на рівні ОС структуру файла, відображаючи кожен файл простою послідовністю байтів. У цьому разі застосування, які працюють із файлами, самі визначають їхній формат.

Такий спрощений підхід справедливий не для всіх файлів. Є спеціальні файли, що їх операційна система інтерпретує особливим чином. Структуру таких файлів ОС підтримує відповідно до тих задач, які з їхньою допомогою розв'язуються.

Ще однією категорією файлів є виконувані файли. Хоч їх звичайно не розглядають разом зі спеціальними файлами, вони мають жорстко заданий формат, який розпізнає операційна система. Часто буває так, що ОС може працювати із виконуваними файлами різних форматів.

Ще одним варіантом класифікації є поділ на *файли із прямим і послідовним доступом*. Файли із прямим доступом дають змогу вільно переходити до будь-якої позиції у файлі, використовуючи для цього поняття *покажчика поточної позиції файла* (seek pointer), що може переміщатися у будь-якому напрямку за допомогою відповідних системних викликів. Файли із послідовним доступом можуть бути зчитані тільки послідовно, із початку в кінець. Сучасні ОС звичайно розглядають усі файли як файли із прямим доступом.

11.1.4. Імена файлів

Важливою складовою роботи із файлами є організація доступу до них за іменем.

Різні системи висувають різні вимоги до імен файлів. Так, у деяких системах імена є чутливими до регістра (myfile.txt і MYFILE.TXT будуть різними іменами), а в інших — ні.

Операційна система може розрізняти окремі частини імені файла. Кілька останніх символів імені (звичайно відокремлені від інших символів крапкою) у деяких системах називають *розширенням файла*, яке може характеризувати його тип. В інших системах обов'язкове розширення не виділяють, при цьому деякі програми можуть, однак, розпізнавати потрібні їм файли за розширеннями (наприклад, компілятор С може розраховувати на те, що вихідні файли програм матимуть розширення .c).

Важливою характеристикою файлової системи є *максимальна довжина імені файла*. У минулому багато ОС різним чином обмежували довжину імен файлів. Широко відоме було обмеження на 8 символів у імені файла і 3 — у розширенні, присутнє у файловій системі FAT до появи Windows 95. Сьогодні стандартним значенням максимальної довжини імені файла є 255 символів.

11.2. Організація інформації у файловій системі

У сучасних ОС файли у файловій системі не прийнято зберігати одним невпорядкованим списком (зазначимо, що можливі винятки, наприклад, для вбудованих систем). Десятки гігабайтів даних, що зберігаються зараз на дисках, вимагають упорядкування, файли, в яких перебувають ці дані, мають бути ефективно організовані. Підходи, що були запропоновані для вирішення цього завдання, наведено нижче.

11.2.1. Розділи

Перед тим, як розглянути підходи до організації інформації безпосередньо у файловій системі, зупинимося на тому, як організовують дисковий простір для розміщення на ньому файлової системи, і введемо поняття *розділу*. Розділи реалізують логічне відображення фізичного диска.

Розділ (partition) — частина фізичного дискового простору, що призначена для розміщення на ній структури однієї файлової системи і з логічної точки зору розглядається як єдине ціле.

Розділ — це логічний пристрій, що з погляду ОС функціонує як окремий диск. Такий пристрій може відповідати всьому фізичному диску (у цьому разі кажуть, що диск містить один розділ); найчастіше він відповідає частині диска (таку частину називають ще фізичним розділом); буває й так, що подібні логічні пристрої поєднують кілька фізичних розділів, що перебувають, можливо, на різних дисках (такі пристрої ще називають *логічними томами* — logical volumes).

Кожний розділ може мати свою файлову систему (і, можливо, використовуватися різними ОС). Для поділу дискового простору на розділи використовують спеціальну утиліту, яку часто називають fdisk. Для генерації файлової системи

на розділі потрібно використати операцію високорівневого форматування диска. У деяких ОС під *томом* (volume) розуміють розділ із встановленою на ньому файловою системою.

Реалізація розділів дає змогу відокремити логічне відображення дискового *простору від фізичного і підвищує гнучкість використання файлових систем.*

11.2.2. Каталоги

Розділи є основою організації великих обсягів дискового простору для розгортання файлових систем. Для організації файлів у рамках розділу зі встановленою файловою системою було запропоновано поняття файлового каталогу (file directory) або просто *каталогу.*

Каталог – це об'єкт (найчастіше реалізований як спеціальний файл), що містить інформацію про набір файлів. Про такі файли кажуть, що вони містяться в каталозі. Файли заносяться в каталоги користувачами на підставі їхніх власних критеріїв, деякі каталоги можуть містити дані, потрібні операційній системі, або її програмний код.

Каталог можна уявити собі як символьну таблицю, що реалізує відображення імен файлів у елементи каталогу (зазвичай в таких елементах зберігають низькорівневу інформацію про файли). Подивимося, як може бути реалізоване таке відображення.

Деревоподібна структура каталогів

Базовою ідеєю організації даних за допомогою каталогів є те, що вони можуть містити інші каталоги. Вкладені каталоги називають підкаталогами (subdirectories). Таким чином формують дерево каталогів. Перший каталог, створений у файловій системі, встановленій у розділі (корінь дерева каталогів), називають кореневим каталогом (root directory).

Поняття шляху

Розглянемо, яким чином формують ім'я файла з урахуванням багаторівневої структури каталогів.

Для файла, розташованого всередині каталогу недостатньо його імені для однозначного визначення, де він перебуває, – в іншому каталозі може бути файл із тим самим ім'ям. Тепер для визначення місцезнаходження файла потрібно додавати до його імені список каталогів, де він перебуває. Такий список називають *шляхом* (path). Каталоги у шляху перераховують зліва направо – від меншої глибини вкладеності до більшої. Роздільник каталогів у шляху відрізняється для різних систем: в UNIX прийнято використовувати прямий слеш «/», а у Windows-системах – зворотний «\».

Абсолютний і відносний шляхи

Є два шляхи до файла: абсолютний і відносний. *Абсолютний* (або повний) повністю й однозначно визначає місце розташування файла. Такий шлях обов'язково має містити кореневий каталог. Ось приклад абсолютного шляху для UNIX-систем: /usr/local/bin/myfile.

Якщо застосування використовує тільки абсолютні шляхи, йому зазвичай бракує гнучкості. Наприклад, у разі перенесення в інший каталог потрібно буде вручну відредагувати всі шляхи, замінивши їх новими.

Відносний — шлях, відлічуваний від деякого місця в ієрархії каталогів. Щоб його організувати, потрібно визначитися із точкою відліку, для чого використовують поняття *поточного каталогу*. Такий каталог задають для кожного процесу, і він може бути змінений у будь-який момент командою cd або системним викликом chdir(). Відносний шлях може відлічуватися від поточного каталогу і звичайно кореневий каталог не включає. Прикладом відносного шляху до файла /usr/ local /bin/myfile (за умови, що поточним каталогом є /usr/local) буде bin/myfile, а в ситуації, коли поточним є каталог файла (/usr/local/bin), відносним шляхом буде просто ім'я файла: myfile.

Для спрощення побудови відносного шляху кожний каталог містить два спеціальні елементи:

- «.», що посилається на поточний каталог;
- «..», що посилається на каталог рівнем вище.

З урахуванням цих елементів можуть бути задані такі відносні шляхи, як ../../bin/myfile (за умови, що поточний каталог — /usr/local/lib/mylib) або ./myfile (вказує на елемент у поточному каталозі).

Застосування, що обмежується тільки відносними шляхами під час доступу до файлів (особливо, якщо вони не виходять за межі каталогу цього застосування), може бути без змін перенесене в інший каталог тієї самої структури.

Є й інші можливості полегшити задання шляхів доступу до файлів у каталогах. Одним із найпоширеніших способів є використання змінної оточення PATH, що містить список часто використовуваних каталогів. У разі доступу до файла за іменем його пошук спочатку виконуватиметься в каталогах, заданих за допомогою PATH.

11.2.3. Зв'язок розділів і структури каталогів

Залишилося з'ясувати важливе питання про взаємозв'язок розділів і структури каталогів файлових систем. Розрізняють два основні підходи до реалізації такого взаємозв'язку, які істотно відрізняються один від одного.

Єдине дерево каталогів. Монтування файлових систем

Перший підхід в основному використовується у файловій системі UNIX і полягає в тому, що розділи зі встановленими на них файловими системами об'єднуються в єдиному дереві каталогів ОС.

Стандартну організацію каталогів UNIX зображують у вигляді дерева з одним коренем — кореневим каталогом, який позначають «/». Файлову систему, на якій перебуває кореневий каталог, називають завантажувальною або кореневою. У більшості реалізацій вона має містити файл із ядром ОС.

Додаткові файлові системи об'єднуються із кореневою за допомогою операції *монтування* (mount). Під час монтування вибраний каталог однієї файлової системи стає кореневим каталогом іншої. Каталог, призначений для монтування файлової системи, називають *точкою монтування* (mount point). Весь вміст файлової

системи, приєднаної за допомогою монтування, виглядає для користувачів системи як набір підкаталогів точки монтування.

Розглянемо операцію монтування на прикладі (рис. 11.1).

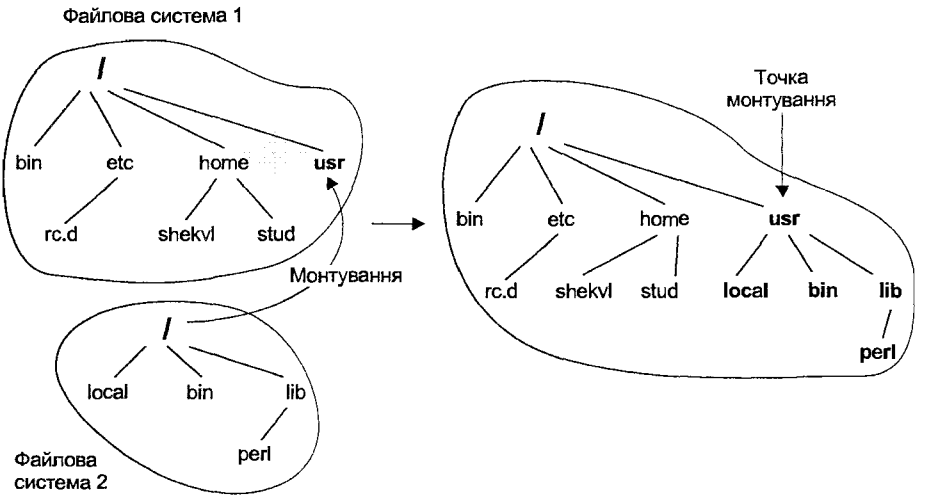

Рис. 11.1. Монтування файлової системи

У цьому разі на диску є два розділи. На кожному з них встановлена файлова система (типи файлових систем можуть бути різними – це не є обмеженням; у каталозі системи одного типу можна змонтувати систему іншого типу за умови, що цей тип підтримує ОС). На рисунку точкою монтування ми вибрали каталог /usr першої файлової системи. Для користувача системи практично не помітно, що насправді каталог / і каталог /usr відповідають різним файловим системам. Відмінності можуть виявлятися, наприклад, під час спроби перенесення файла: виконання звичайної операції перенесення (mv у UNIX) між файловими системами не дозволяється.

Розглянемо деякі наслідки застосування єдиного каталогу для організації файлової системи.

- Будь-який файл може бути адресований побудовою відносного шляху від будь-якого каталогу.
- Від користувача прихована структура розділів жорсткого диска, яка йому у більшості випадків не потрібна.
- Адміністрування системи спрощується. Наприклад, якщо додамо ще один диск і захочемо перенести на нього каталог /home, достатньо буде виконати кілька простих дій: відформатувати цей диск, задавши на ньому один розділ; змонтувати цей розділ у довільному місці; перенести на нього каталог /home (стерши весь його вміст на вихідному диску); заново змонтувати цей розділ у каталозі /home кореневої файлової системи.

Внаслідок цих дій всі застосування, які використовують каталог /home, працюватимуть у колишньому режимі; на їхню роботу не вплине той факт, що каталог тепер відповідає новій файловій системі, а /home став точкою монтування.

Літерні позначення розділів

Другий підхід, що в основному поширений в лініях Consumer Windows і Windows XP, припускає, що кожний розділ зі встановленою файловою системою є видимим для користувача і позначений буквою латинського алфавіту. Такий розділ звичайно називають томом. Позначення томів нам знайомі — це C:, D: тощо.

Особливості такої реалізації наведені нижче.

- Вміст кожного розділу не пов'язаний з іншими розділами; відносний шлях можна побудувати тільки за умови, що поточний каталог перебуває на тому самому томі, що і файл.
- Структура логічних розділів видима для користувача.
- Перенос каталогу на новий розділ призводить до того, що шлях до цього каталогу зміниться (оскільки такий шлях завжди включає літерне позначення тому). У підсумку програмне забезпечення, яке використовує цей шлях, може перестати працювати.
- У разі необхідності додавання або вилучення дискового пристрою у системах лінії Consumer Windows користувач не може впливати на те, які літери система присвоює розділам (фактично це залежить від порядку підключення апаратних пристроїв); у системах лінії Windows XP користувач може вільно змінювати літерні позначення під час роботи системи.

Зазначимо, що нині в ОС лінії Windows XP реалізована підтримка монтування (для файлової системи NTFS), що вирішує більшість перелічених проблем. Ця підтримка вперше з'явилась у Windows 2000 [70].

11.3. Зв'язки

Структура каталогів файлової системи не завжди є деревом. Багато файлових систем дає змогу задавати кілька імен для одного й того самого файла. Такі імена називають *зв'язками* (links). Розрізняють жорсткі та символічні зв'язки.

11.3.1. Жорсткі зв'язки

Ім'я файла не завжди однозначно пов'язане з його даними. За підтримки жорстких зв'язків (hard links) для файла допускається кілька імен. Усі жорсткі зв'язки визначають одні й ті самі дані на диску, для користувача вони не відрізняються: не можна визначити, які з них були створені раніше, а які — пізніше.

Підтримка жорстких зв'язків у POSIX

Для створення жорстких зв'язків у POSIX призначений системний виклик `link()`. Першим параметром він приймає ім'я вихідного файла, другим — ім'я жорсткого зв'язку, що буде створений:

```
#include <unistd.h>  // для стандартних файлових операцій POSIX
link ("myfile.txt", "myfile-hardlink.txt");
```

Зазначимо, що стандартні засоби вилучення даних за наявності жорстких зв'язків працюватимуть саме з ними, а не безпосередньо із файлами. Замість системного

виклику вилучення файла використовують виклик вилучення зв'язку (який зазвичай називають `unlink()`), що вилучатиме один жорсткий зв'язок для заданого файла. Якщо після цього зв'язків у файла більше не залишається, його дані також вилучаються.

```
// вилучити файл, якщо в нього був один жорсткий зв'язок
unlink("myfile.txt");
```

Підтримка жорстких зв'язків у Windows XP

Жорсткі зв'язки здебільшого реалізовані в UNIX-сумісних системах, їх підтримують також у системах лінії Windows XP для файлової системи NTFS [69]. Для створення жорсткого зв'язку в цій системі необхідно використати функцію `CreateHardLink()`, ім'я зв'язку задають першим параметром, ім'я файла – другим, а третій дорівнює нулю:

```
CreateHardLink("myfile_hardlink.txt", "myfile.txt", 0);
```

Для вилучення жорстких зв'язків у Win32 API використовують функцію `DeleteFile()`:

```
DeleteFile("myfile_hardlink.txt");
```

Зазначимо, що для файлових систем, які не підтримують жорстких зв'язків, виклик `DeleteFile()` завжди спричиняє вилучення файла.

Жорсткі зв'язки мають певні недоліки, які обмежують їх застосування:

- не можуть бути задані для каталогів;
- усі жорсткі зв'язки одного файла завжди мають перебувати на одному й тому самому розділі жорсткого диска (в одній файловій системі);
- вилучення жорсткого зв'язку потенційно може спричинити втрати даних файла.

11.3.2. Символічні зв'язки

Основні поняття

Символічний зв'язок (symbolic link) – зв'язок, фізично відокремлений від даних, на які вказує. Фактично, це спеціальний файл, що містить ім'я файла, на який вказує.

Наведемо властивості символічних зв'язків.

- Через такий зв'язок здійснюють доступ до вихідного файла.
- При вилученні зв'язку, вихідний файл не зникне.
- Якщо вихідний файл перемістити або вилучити, зв'язок розірветься, і доступ через нього стане неможливий, якщо файл потім поновити на тому самому місці, зв'язком знову можна користуватися.
- Символічні зв'язки можуть вказувати на каталоги і файли, що перебувають на інших файлових системах (на іншому розділі жорсткого диска). Наприклад, якщо створити в поточному каталозі зв'язок system-docs, що вказує на каталог /usr/doc, то перехід у каталог system-docs призведе до переходу в каталог /usr/doc.

Підтримка символічних зв'язків на рівні системних викликів

Для задання символічного зв'язку у POSIX визначено системний виклик `symlink()`, параметри якого аналогічні до параметрів `link()`:

```
symlink ("myfile.txt", "myfile-symlink.txt");
```

Для отримання шляху до файла або каталогу, на який вказує символічний зв'язок, використовують системний виклик `readlink()`.

```
// PATH_MAX – константа, що задає максимальну довжину шляху
char filepath[PATH_MAX+1];
readlink("myfile-symlink.txt", filepath, sizeof(filepath));
// у filepath буде шлях до myfile.txt
```

Символічні зв'язки вперше з'явилися у файлових системах UNIX, у Windows XP вони підтримуються файловою системою NTFS під назвою *точок з'єднання* (junction points), але засоби API для їхнього використання не визначені [87].

11.4. Атрибути файлів

Кожний файл має набір характеристик — *атрибутів*. Набір атрибутів змінюється залежно від файлової системи. Найпоширеніші атрибути файла наведено нижче.

- Ім'я файла, докладно розглянуте раніше.
- Тип файла, який звичайно задають для спеціальних файлів (каталогів, зв'язків тощо).
- Розмір файла (зазвичай для файла можна визначити його поточний, а іноді й максимальний розмір).
- Атрибути безпеки, що визначають права доступу до цього файла (про такі атрибути йтиметься в розділі 18).
- Часові атрибути, до яких належать час створення останньої модифікації та останнього використання файла.

Інформацію про атрибути файла також зберігають на диску. Особливості її зберігання залежать від фізичної організації файлової системи.

11.5. Операції над файлами і каталогами

У цьому розділі вивчатимемо основні операції, які можна виконувати над файлами та каталогами.

11.5.1. Підходи до використання файлів процесами

Підходи до використання файлів із процесу бувають такі: *зі збереженням* (stateful) і *без збереження стану* (stateless).

У разі збереження стану є спеціальні операції, які готують файл до використання у процесі (*відкривають його*) і скасовують цю готовність (*закривають його*). Інші операції використовують структури даних, підготовлені під час відкриття

файла, і можуть виконуватися тільки доти, поки файл не буде закритий. Перевагою такого підходу є висока продуктивність, оскільки під час відкриття файла потрібні структури даних завантажуються у пам'ять.

Якщо стан не зберігають, кожна операція роботи із файлом (читання, записування тощо) супроводжується повною підготовкою файла до роботи (кожна операція починається відкриттям файла і завершується закриттям). Хоча такий підхід програє у продуктивності, його можна використати для підвищення надійності роботи системи за високої ймовірності того, що файлова операція зазнає краху, внаслідок чого структури даних відкритих файлів залишаться в некоректному стані. Так можна робити у випадку, коли файлову систему використовують через мережу, тому що у будь-який момент може статися розрив мережного з'єднання.

Далі в цьому розділі буде розглянуто підхід зі збереженням стану.

11.5.2. Загальні відомості про файлові операції

Назвемо основні файлові операції, які звичайно надає операційна система для використання у прикладних програмах.

- **Відкриття файла.** Після відкриття файла процес може із ним працювати (наприклад, робити читання і записування). Відкриття файла зазвичай передбачає завантаження в оперативну пам'ять спеціальної структури даних — *дескриптора файла*, який визначає його атрибути та місце розташування на диску. Наступні виклики використовуватимуть цю структуру для доступу до файла.
- **Закриття файла.** Після завершення роботи із файлом його треба закрити. При цьому структуру даних, створену під час його відкриття, вилучають із пам'яті. Усі дотепер не збережені зміни записують на диск.
- **Створення файла.** Ця операція спричиняє створення на диску нового файла нульової довжини. Після створення файл автоматично відкривають.
- **Вилучення файла.** Ця операція спричиняє вилучення файла і вивільнення зайнятого ним дискового простору. Вона зазвичай недопустима для відкритих файлів. У розділі 11.3 зазначалося про особливості реалізації цієї операції у системі з підтримкою жорстких зв'язків.
- **Читання з файла.** Ця операція звичайно зводиться до пересилання певної кількості байтів із файла, починаючи із поточної позиції, у заздалегідь виділений для цього буфер пам'яті режиму користувача.
- **Записування у файл.** Здійснюють із поточної позиції, дані записують у файл із заздалегідь виділеного буфера. Якщо на цій позиції вже є дані, вони будуть перезаписані. Ця операція може змінити розмір файла.
- **Переміщення покажчика поточної позиції.** Перед операціями читання і записування слід визначити, де у файлі перебувають потрібні дані або куди треба їх записати, задавши за допомогою цієї операції поточну позицію у файлі. Зазначимо, що якщо перемістити покажчик файла за його кінець, а потім виконати операцію записування, довжина файла збільшиться.
- **Отримання і задання атрибутів файла.** Ці дві операції дають змогу зчитувати поточні значення всіх або деяких атрибутів файла або задавати для них нові значення.

11.5.3. Файлові операції POSIX

Усі UNIX-системи реалізують доступ до файлів за допомогою компактного набору системних викликів, визначеного стандартом POSIX, який відповідає набору файлових операцій, наведених у попередньому розділі.

Відкриття і створення файлів

Для відкриття файла використовують системний виклик open(), першим параметром якого є шлях до файла.

```
#include <fcntl.h>
int open(const char *pathname, int flags[, mode_t mode]);
```

Виклик open() повертає цілочислове значення — *файловий дескриптор*. Його слід використовувати в усіх викликах, яким потрібен відкритий файл. Природа дескриптора файла в Linux буде описана у розділі 13.1.4. У разі помилки цей виклик поверне –1, а значення змінної errno відповідатиме коду помилки.

Розглянемо деякі значення, яких може набувати параметр flags (їх можна об'єднувати за допомогою побітового «або»):

- O_RDONLY, O_WRONLY, O_RDWR — відкриття файла, відповідно, тільки для читання, тільки для записування або для читання і записування (має бути задане одне із цих трьох значень, наведені нижче не обов'язкові);
- O_CREAT — якщо файл із таким ім'ям відсутній, його буде створено, якщо файл є і увімкнуто прапорець O_EXCL, буде повернено помилку;
- O_TRUNC — якщо файл відкривають для записування, його довжину покладають рівною нулю;
- O_NONBLOCK — задає *неблокувальне введення-виведення;* особливості його використання розглянемо разом із викликом read().

Параметр mode потрібно задавати тільки тоді, коли задано прапорець O_CREAT. Значенням у цьому випадку буде вісімкове число, що задає права доступу до файла. Докладно ці права буде розглянуто в розділі 18, а поки що задаватимемо як аргумент значення 0644, що дає змогу після створення файла записувати в нього дані.

Ось приклад використання цього системного виклику:

```
// відкриття файла для записування
int fd1 = open("./myfile.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);
// відкриття файла для читання, помилка, якщо файла немає
int fd1 = open("./myfile.txt", O_RDONLY);
```

Закриття файла

Файл закривають за допомогою системного виклику close(), що приймає файловий дескриптор:

```
close(fd1);
```

Читання і записування даних

Для читання даних із відкритого файла використовують системний виклик read():

```
ssize_t read(int fd1, void *buf, size_t count);
```

Внаслідок цього виклику буде прочитано count байтів із файла, заданого відкритим дескриптором fd1, у пам'ять, на яку вказує buf (ця пам'ять виділяється

заздалегідь). Виклик read() повертає реальний обсяг прочитаних даних (тип ssize_t є цілочисловим). Покажчик позиції у файлі пересувають за зчитані дані.

```
char buf[100];
// читають 100 байт з файлу в buf
int bytes_read = read(fd1, buf, sizeof(buf));
```

Коли потрібна кількість даних у конкретний момент відсутня (наприклад, fd1 пов'язаний із мережним з'єднанням, яким ще не прийшли дані), поведінка цього виклику залежить від значення прапорця O_NONBLOCK під час виклику open().

У разі блокувального виклику (O_NONBLOCK не увімкнуто) він призупинить поточний потік до тих пір, поки дані не з'являться, а в разі неблокувального (прапорець O_NONBLOCK увімкнуто) — зчитає всі доступні дані й завершиться, призупинення потоку не відбудеться.

Для записування даних у відкритий файл через файловий дескриптор використовують системний виклик write():

```
ssize_t write(int fd1, const void *buf, size_t count);
```

Внаслідок цього виклику буде записано count байтів у файл через дескриптор fd1 із пам'яті, на яку вказує buf. Виклик write() повертає обсяг записаних даних.

```
int fd1, bytes_written;
fd1 = open("./myfile.txt", O_RDWR|O_CREAT, 00644);
bytes_written = write(fd1, "hello", sizeof("hello");
```

Реалізація копіювання файлів

Наведемо приклад реалізації копіювання файлів за допомогою засобів POSIX.

```
char buf[1024]; int bytes_read, intfile, outfile;
// відкриття вихідного файла для читання
infile = open("infile.txt", O_RDONLY);
if (infile == -1) {
    printf ("помилка під час відкриття файла\n"); exit(-1);
}
// створення результуючого файла, перевірку помилок пропущено
outfile = open("outfile.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);
do {
    // читання даних із вихідного файла у буфер
    bytes_read = read(infile, buf, sizeof(buf));
    // записування даних із буфера в результуючий файл
    if (bytes_read > 0) write(outfile, buf, bytes_read);
} while (bytes_read > 0);
// закриття файлів
close(infile);
close(outfile);
```

Переміщення покажчика поточної позиції у файлі

Кожному відкритому файлу відповідає покажчик позиції (зсув) усередині файла. Його можна пересувати за допомогою системного виклику lseek():

```
off_t lseek(int fd1, off_t offset, int whence);
```

Параметр offset задає величину переміщення покажчика. Режим переміщення задають параметром whence, який може набувати значень SEEK_SET (абсолютне

переміщення від початку файла), SEEK_CUR (відносне переміщення від поточного місця покажчика позиції) і SEEK_END (переміщення від кінця файла).

```
// переміщення покажчика позиції на 100 байт від поточного місця
lseek (outfile, 100, SEEK_CUR);
// записування у файл
write (outfile, "hello", sizeof("hello"));
```

Коли покажчик поточної позиції перед операцією записування опиняється за кінцем файла, він внаслідок записування автоматично розширюється до потрібної довжини. На цьому ґрунтується ефективний спосіб створення файлів необхідного розміру:

```
int fd1 = open("file", O_RDWR|O_CREAT|O_TRUNC, 0644); // створення файла
lseek(fd1, needed_size, SEEK_SET);                    // розширення до потрібного розміру
write(fd1, "", 1);                                    // записування нульового байта
```

Збирання інформації про атрибути файла

Для отримання інформації про атрибути файла (тобто про вміст його індексного дескриптора) використовують системний виклик stat().

```
#include <sys/stat.h>
int stat(const char *path, struct stat *attrs);
```

Першим параметром є шлях до файла, другим — структура, у яку записуватимуться атрибути внаслідок виклику. Деякі поля цієї структури (всі цілочислові) наведено нижче:

- st_mode — тип і режим файла (бітова маска прапорців, зокрема прапорець S_IFDIR встановлюють для каталогів);
- st_nlink — кількість жорстких зв'язків;
- st_size — розмір файла у байтах;
- st_atime, st_mtime, st_ctime — час останнього доступу, модифікації та зміни атрибутів (у секундах з 1 січня 1970 року).

Ось приклад відображення інформації про атрибути файла:

```
struct stat attrs;
stat("myfile", &attrs);
if (attrs.st_imode & S_IFDIR)
    printf("myfile є каталогом\n");
else printf("розмір файла: %d\n", attrs.st_size);
```

Для отримання такої самої інформації з дескриптора відкритого файла використовують виклик fstat().

```
int fstat(int fd1, struct stat *attrs);
```

11.5.4. Файлові операції Win32 API

Win32 API містить набір функцій для роботи з файлами, багато в чому аналогічних до системних викликів POSIX. Зупинимося на цьому наборі.

Відкриття і створення файлів

Аналогом системного виклику open() у Win32 API є функція CreateFile():

```
HANDLE CreateFile (LPCTSTR fname, DWORD amode, DWORD smode,
  LPSECURITY_ATTRIBUTES attrs, DWORD cmode, DWORD flags, HANDLE tfile);
```

Першим параметром є ім'я файла. Параметр amode задає режим відкриття файла і може набувати значень GENERIC_READ (читання) і GENERIC_WRITE (записування). Параметр smode задає можливість одночасного доступу до файла: 0 означає, що доступ неможливий. Параметр cmode може набувати таких значень:

- CREATE_NEW — якщо файл є, повернути помилку, у протилежному випадку створити новий;
- CREATE_ALWAYS — створити новий файл, навіть якщо такий уже є;
- OPEN_EXISTING — якщо файл є, відкрити його, якщо немає, повернути помилку;
- OPEN_ALWAYS — якщо файл є, відкрити його, у протилежному випадку створити новий.

Під час створення файла значенням параметра flags може бути FILE_ATTRIBUTE_NORMAL, що означає створення файла зі стандартними атрибутами.

Ця функція повертає дескриптор відкритого файла. У разі помилки буде повернуто певне значення INVALID_HANDLE_VALUE, рівне –1.

```
// відкриття наявного файла
HANDLE infile = CreateFile("infile.txt", GENERIC_READ, 0, NULL,
    OPEN_EXISTING, 0, 0);
// створення нового файла
HANDLE outfile = CreateFile("outfile.txt", GENERIC_WRITE, 0, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
```

Керування успадкуванням файлових дескрипторів

На відміну від POSIX, у Win32 відкриті файлові дескриптори за замовчуванням не успадковуються нащадками цього процесу, навіть якщо під час виклику CreateProcess() параметр inherit_handles встановлений у TRUE (див. розділ 3.10.6). Потрібно додатково дозволити таке успадкування для кожного дескриптора при його відкритті. Це забезпечують установкою поля bInheritHandle (третього один по одному) структури атрибутів безпеки, покажчик на яку передають у CreateFile(). Зауважимо, що першим полем структури є її розмір, а друге описує права доступу для об'єкта і до розділу 18 буде встановлюватися в NULL.

```
SECURITY_ATTRIBUTES sa = { sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
HANDLE fh = CreateFile("outfile.txt", GENERIC_WRITE, 0, &sa, ...);
```

Закриття файлів

Для закриття дескриптора файла застосовують функцію CloseHandle():

```
CloseHandle(infile);
```

Читання і записування даних

Для читання з файла використовують функцію ReadFile():

```
BOOL ReadFile( HANDLE fh, LPCVOID buf, DWORD len,
               LPDWORD pbytes_read, LPOVERLAPPED over );
```

Параметр buf задає буфер для розміщення прочитаних даних, len — кількість байтів, які потрібно прочитати, за адресою pbytes_read буде збережена кількість прочитаних байтів (коли під час спроби читання трапився кінець файла, *pbytes_read

не дорівнюватиме len). Виклик ReadFile() поверне TRUE у разі успішного завершення читання.

```
char buf[100]; DWORD bytes_read;
ReadFile(outfile, buf, sizeof(buf), &bytes_read, 0);
if (bytes_read != sizeof(buf))
    printf("Досягнуто кінця файла\n");
```

Для записування у файл використовують функцію WriteFile():

```
BOOL WriteFile( HANDLE fh, LPCVOID buf, DWORD len,
                LPDWORD pbytes_written, LPOVERLAPPED over );
```

Параметр buf задає буфер, з якого йтиме записування, len — обсяг записуваних даних, за адресою pbytes_written буде збережена кількість записаних байтів. Виклик WriteFile() поверне TRUE, якщо записування завершене успішно.

```
DWORD bytes_written;
WriteFile(outfile, "hello", sizeof("hello"), &bytes_written, 0);
```

Реалізація копіювання файлів

Наведемо приклад реалізації копіювання файлів за допомогою засобів Win32 API.

```
char buf[1024]; DWORD bytes_read, bytes_written;
HANDLE infile = CreateFile("infile.txt", GENERIC_READ, 0, 0,
   OPEN_EXISTING, 0, 0);
if (infile == INVALID_HANDLE_VALUE) {
    printf("Помилка під час відкриття файла\n"); exit(-1);
}
HANDLE outfile = CreateFile("outfile.txt", GENERIC_WRITE, 0, 0,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
do {
    ReadFile(infile, buf, sizeof(buf), &bytes_read, 0);
    if (bytes_read > 0)
        WriteFile(outfile, buf, bytes_read, &bytes_written, 0);
} while (bytes_read > 0);
CloseHandle(infile);
CloseHandle(outfile);
```

Прямий доступ до файла

Прямий доступ до файла реалізований функцією SetFilePointer():

```
DWORD SetFilePointer(HANDLE fh, LONG offset,
                     PLONG offset_high, DWORD whence);
```

Параметри offset і whence аналогічні до однойменних параметрів lseek(). Константи режиму переміщення визначені як FILE_BEGIN (аналогічно до SEEK_SET), FILE_CURRENT (аналогічно до SEEK_CUR) і FILE_END (аналогічно до SEEK_END).

```
// переміщення покажчика позиції на 100 байт від початку файла
SetFilePointer(outfile, 100, NULL, FILE_BEGIN);
// записування у файл
WriteFile(outfile, "hello", sizeof("hello"), &bytes_written, 0);
```

Створення файла заданої довжини виконують аналогічно до прикладу для системного виклику lseek().

Збирання інформації про атрибути файла

Атрибути файла із заданим ім'ям можуть бути отримані за допомогою функції GetFileAttributesEx(). Вона заповнює структуру WIN32_FILE_ATTRIBUTE_DATA з полями, аналогічними stat():

- dwFileAttributes — маска прапорців (зокрема, для каталогів задають прапорець FILE_ATTRIBUTE_DIRECTORY);
- ftCreationTime, ftLastAccessTime, ftLastWriteTime — час створення, доступу і модифікації файла (структури типу FILETIME);
- nFileSizeLow — розмір файла (якщо для його відображення не достатньо 4 байт, додатково використовують поле nFileSizeHigh).

```
WIN32_FILE_ATTRIBUTE_DATA attrs;
// другий параметр завжди задають однаково
GetFileAttributesEx("myfile", GetFileExInfoStandard, &attrs);
if (attrs.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
    printf("myfile є каталогом\n");
else printf("розмір файла: %d\n", attrs.nFileSizeLow);
```

Для отримання доступу до атрибутів відкритого файла за його дескриптором використовують функцію GetFileInformationByHandle(). Вона дає змогу отримати повнішу інформацію із файла (зокрема, кількість його жорстких зв'язків).

Є також функції, що повертають значення конкретних атрибутів:

```
long size = FileSize(fh);   // розмір файла за його дескриптором
```

11.5.5. Операції над каталогами

Розглянемо базові операції над каталогами.

- **Створення нового каталогу.** Ця операція створює новий каталог. Він звичайно порожній, деякі реалізації автоматично додають у нього елементи «.» і «..».
- **Вилучення каталогу.** На рівні системного виклику ця операція дозволена тільки для порожніх каталогів.
- **Відкриття і закриття каталогу.** Каталог, подібно до звичайного файла, має бути відкритий перед використанням і закритий після використання. Деякі операції, пов'язані із доступом до елементів, допустимі тільки для відкритих каталогів.
- **Читання елемента каталогу.** Ця операція зчитує один елемент каталогу і переміщує поточну позицію на наступний елемент. Використовуючи читання елемента каталогу в циклі, можна обійти весь каталог.
- **Перехід у початок каталогу.** Ця операція переміщує поточну позицію до першого елемента каталогу.

Робота з каталогами в POSIX

Для створення каталогу використовують виклик mkdir(), що приймає як параметр шлях до каталогу і режим.

```
if (mkdir("./newdir", 0644) == -1)
    printf("помилка під час створення каталогу\n");
```

Вилучення порожнього каталогу за його іменем відбувається за допомогою виклику rmdir():

```
if (rmdir("./dir") == -1)
    printf("помилка у разі вилучення каталогу\n");
```

Відкривають каталог викликом opendir(), що приймає як параметр ім'я каталогу:

```
DIR *opendir(const char *dirname);
```

Під час виконання opendir() ініціалізується внутрішній покажчик поточного елемента каталогу. Цей виклик повертає *дескриптор каталогу* — покажчик на структуру типу DIR, що буде використана під час обходу каталогу. У разі помилки повертають NULL.

Для читання елемента каталогу і переміщення внутрішнього покажчика поточного елемента використовують виклик readdir():

```
struct dirent *readdir(DIR *dirp);
```

Цей виклик повертає покажчик на структуру dirent, що описує елемент каталогу (із полем d_name, яке містить ім'я елемента) або NULL, якщо елементів більше немає.

Після закінчення пошуку потрібно закрити каталог за допомогою виклику closedir(). Якщо необхідно перейти до першого елемента каталогу без його закриття, використовують виклик rewinddir(). Обидва ці виклики приймають як параметр дескриптор каталогу.

Наведемо приклад коду обходу каталогу в POSIX.

```
DIR *dirp; struct dirent *dp;
dirp = opendir("./dir");
if (! dirp) { printf("помилка під час відкриття каталогу\n"); exit(-1); }
while (dp = readdir(dirp)) {
    printf ("%s\n", dp->d_name);  // відображення імені елемента
}
closedir (dirp);
```

Робота з каталогами у Win32 API

Для створення каталогу використовують функцію CreateDirectory(), що приймає як параметри шлях до каталогу та атрибути безпеки.

```
if (! CreateDirectory("c:\\newdir", 0))
    printf("помилка під час створення каталогу\n");
```

Вилучення порожнього каталогу за його іменем відбувається за допомогою функції RemoveDirectory(). Якщо каталог непорожній, ця функція не вилучає його і повертає FALSE.

```
if (! RemoveDirectory("c:\\dir"))
    printf("помилка у разі вилучення каталогу\n");
```

Відкривають каталог функцією FindFirstFile():

```
HANDLE FindFirstFile(LPCSTR path, LPWIN32_FIND_DATA fattrs);
```

Параметр path задає набір файлів. Він може бути ім'ям каталогу (в набір входитимуть усі файли цього каталогу), крім того, у ньому допустимі символи шаблону «*» і «?». Параметр fattrs — це покажчик на структуру, що буде заповнена інформацією про знайдений файл. Структура подібна до WIN32_FILE_ATTRIBUTE_DATA, але в ній додатково зберігають ім'я файла (поле cFileName).

Ця функція повертає *дескриптор пошуку*, який можна використати для подальшого обходу каталогу. Для доступу до такого файла в каталозі використовують функцію FindNextFile(), у яку передають дескриптор пошуку і таку саму структуру, як у FindFirstFile():

```
BOOL FindNextFile(HANDLE findh, LPWIN32_FIND_DATA fattrs);
```

Якщо файлів більше немає, ця функція повертає FALSE.

Після закінчення пошуку потрібно закрити дескриптор пошуку за допомогою FindClose(). CloseHandle() для цього використати не можна.

Наведемо приклад коду обходу каталогу в Win32 API:

```
WIN32_FIND_DATA fattrs;
HANDLE findh = FindFirstFile("c:\\mydir\\*", &fattrs);
do {
    printf ("%s\n", fattrs.cFileName);  // відображення імені елемента
} while (FindNextFile(findh, &fattrs));
FindClose(findh);
```

11.6. Міжпроцесова взаємодія на основі інтерфейсу файлової системи

Як зазначалося, використання файлів є найпростішою формою міжпроцесової взаємодії. У цьому розділі вивчатимемо складніші та продуктивніші способи обміну даними між процесами, які також ґрунтуються на інтерфейсі файлової системи, серед них механізм синхронізації (файлові блокування), реалізація відображуваної і розподілюваної пам'яті та механізм обміну повідомленнями (поіменовані канали).

11.6.1. Файлові блокування

Файлові блокування (file locks) є засобом синхронізації процесів, які намагаються здійснити доступ до одного й того самого файла. Процес може заблокувати файл повністю або будь-який його діапазон (аж до одного байта), після чого інші процеси не зможуть отримати доступу до цього файла або діапазону доти, поки з нього не буде зняте блокування.

Розрізняють *консультативне*, або *кооперативне* (advisory lock), і *обов'язкове блокування* (mandatory lock) файлів.

Консультативне блокування є основним, найбезпечнішим видом блокування. Його підтримують на рівні процесів режиму користувача. Для коректної синхронізації всі процеси перед доступом до файла мають перевіряти наявність такого блокування (якщо блокування відсутнє, процес запроваджує своє блокування, виконує дії із файлом і знімає блокування). Якщо процес виконає операцію читання із файла або записування у файл без попередньої перевірки консультативного блокування, система дозволить виконання цього виклику.

Обов'язкове блокування підтримують на рівні ядра. Коли процес запровадив обов'язкове блокування, жодні операції над файлом або його діапазоном не будуть можливими доти, поки це блокування не буде зняте. Насправді таке блокування може бути небезпечним, оскільки навіть користувач із правами адміністратора не може його зняти (так, випадкове блокування системного файла може зробити систему непрацездатною).

Для файлових блокувань, аналогічно до блокувань читання-записування з розділу 5, розрізняють режими для читання і для записування.

Файлові блокування POSIX

Підтримка файлових блокувань у POSIX грунтується на системному виклику `fcntl()`, що дає змогу запровадити або перевірити блокування на файл або на діапазон даних усередині файла [24, 52].

```
#include <fcntl.h>
int fcntl(int fd1, int cmd, struct flock *lock);
```

Значеннями параметра `cmd` може бути `F_GETLK` — перевірити, чи є блокування, `F_SETLK` — запровадити блокування, якщо воно вже є, повернути помилку, `F_SETLKW` — запровадити блокування, перейти до очікування, якщо воно вже є.

Структура `flock` має бути задана перед викликом. У ній можна вказати:

- діапазон байтів у файлі (поля `l_start` і `l_end`); якщо ці поля дорівнюють нулю, блокується весь файл;
- тип блокування (поле `l_type` із можливими значеннями `F_RDLCK` — для читання, `F_WRLCK` — для записування, `F_UNLCK` — зняти блокування).

Якщо `cmd` дорівнює `F_GETLK`, цю структуру заповнюють усередині виклику, її поле `l_type` міститиме тип блокування, якщо воно є.

```
fd1 = open("lockfile", O_WRONLY|O_CREAT);
// задати структуру flock
struct flock lock = {0};
// задати блокування для записування
lock.l_type = F_WRLCK;
fcntl(fd1, F_SETLKW, &lock);
// зняти блокування
lock.l_type = F_UNLCK;
fcntl(fd1, F_SETLKW, &lock);
close(fd1);
```

За замовчуванням таке блокування є консультативним, для запровадження обов'язкового блокування потрібно спочатку задати спеціальні права доступу до файла (задати setgid біт і очистити дозвіл виконання для групи — див. розділ 18), після чого застосування `fcntl()` до цього файла спричиняє обов'язкове блокування.

Файлові блокування використовують у UNIX-системах як простий і надійний засіб синхронізації процесів. Зазвичай для цієї мети створюють окремий файл блокування, який блокується процесами у разі необхідності доступу до спільно використовуваних ресурсів і вивільняється разом із цими ресурсами. Наприклад, так можна заборонити повторний запуск уже запущеного застосування.

Файлові блокування у Win32

У Windows XP для запровадження обов'язкових файлових блокувань використовують функцію LockFileEx(), а для консультативних — LockFile() [31]. Зупинимось на особливостях використання LockFileEx().

```
BOOL LockFileEx (HANDLE fh, DWORD flags, DWORD dummy, DWORD lcount,
    DWORD hcount, LPOVERLAPPED ov);
```

де: fh — дескриптор відкритого файла;

flags — прапорці режиму блокування, зокрема LOCKFILE_EXCLUSIVE_LOCK — блокування для записування (якщо це значення не задане, встановлюють блокування для читання), LOCKFILE_FAIL_IMMEDIATELY — повернути нуль негайно, якщо файл уже заблокований;

lcount — кількість заблокованих байтів;

ov — покажчик на структуру типу OVERLAPPED, поле Offset якої визначає зсув заблокованої ділянки від початку файла.

LockFileEx() повертає нуль, якщо блокування запровадити не вдалося.

Для розблокування використовують функцію UnLockFileEx() з тими самими параметрами, за винятком flags.

```
HANDLE fh = CreateFile( "lockfile", GENERIC_WRITE, ... );
OVERLAPPED ov = { 0 };
long size = FileSize(fh);
// задати блокування всього файла для записування
LockFileEx(fh, LOCKFILE_EXCLUSIVE_LOCK, 0, size, 0, &ov);
// зняти блокування
UnLockFileEx(fh, 0, size, 0, &ov);
CloseHandle (fh);
```

11.6.2. Файли, що відображаються у пам'ять

Цей розділ буде присвячено реалізації відображуваної пам'яті на основі інтерфейсу файлової системи. Після відображення доступ до такої пам'яті спричиняє виконання операцій доступу до вмісту відображеного файла. Таку технологію широко застосовують у сучасних операційних системах — в UNIX вона визначена стандартом POSIX [24, 37, 52], у лінії Windows XP її реалізація доступна через Win32 API [31, 50].

Принципи дії відображуваної пам'яті. Інтерфейс відображуваної пам'яті POSIX

Відображення файлів у пам'ять відбувається за допомогою спеціального системного виклику; у POSIX такий виклик визначений як mmap(). Він призводить до того, що у визначену частину адресного простору процесу відображають заданий файл або його частину. Після виконання цього системного виклику доступ до такої пам'яті спричинятиме прямий доступ до вмісту цього файла (читання пам'яті аналогічне до читання із файла, її зміна аналогічна до зміни файла). Перед використанням цього виклику файл має бути відкритий.

Відображення в цьому разі працює так. Коли файл відобразився на ділянку пам'яті, що починається з адреси *p*, то:

- у разі доступу до байта пам'яті за адресою *p* буде отримано нульовий (початковий) байт цього файла;
- у разі зміни байта пам'яті за адресою *p*+(*N* байт) буде змінено *N*-й байт файла (починаючи від 0);
- доступ до пам'яті за кінцем файла неможливий і призводить до помилки.

У разі закриття файла або завершення роботи процесу модифіковану інформацію зберігають у файлі на диску.

Загальний принцип відображення файла в адресний простір процесу показано на рис. 11.2.

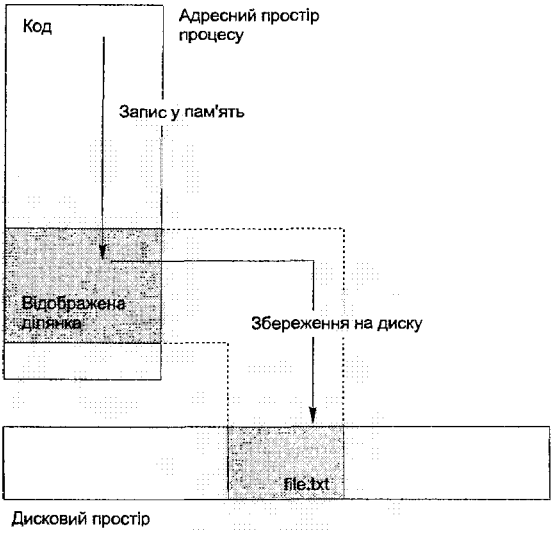

Рис. 11.2. Файл, що відображається у пам'ять

Системний виклик `mmap()` відповідно до POSIX має такий синтаксис:

```
#include <sys/mman.h>
void *mmap (void *start, size_t len, int prot, int flags, int fdl,
            off_t offset);
```

де: `start` — адреса, з якої почнеться відображувана ділянка (найчастіше цей параметр покладають рівним `NULL`, надаючи ОС самій визначити, за якою адресою почати відображення);

`len` — розмір відображуваної ділянки у байтах;

`prot` — режим доступу до відображуваної пам'яті (`PROT_READ` — дозволене читання, `PROT_WRITE` — дозволене записування);

`flags` — прапорці відображення (`MAP_SHARED` — усі зміни мають бути негайно записані у відображуваний файл без буферизації, `MAP_PRIVATE` — під час записування в ділянку буде створено копію відображуваного файла, і подальші зміни відбуватимуться з нею);

`fd1` – дескриптор відображуваного файла;

`offset` – зсув у файлі, із якого почнеться відображення.

Результатом виклику `mmap()` буде адреса, з якої почате відображення.

Розрив відображення трапляється у разі використання системного виклику, що у POSIX визначений як `munmap()`, після чого відповідна ділянка пам'яті більше не буде пов'язана із файлом, і спроба доступу до неї спричинить помилку. Розрив відображення теж призводить до збереження інформації у відповідному файлі.

```
int munmap(void *start, size_t len);
```

При завершенні програми необхідності явного виклику `munmap()` немає – записування змін на диск виконуватиметься автоматично.

У разі використання відображуваної пам'яті необхідно виконати такі дії.

1. Відкрити файл із режимом, відповідним до того, як потрібно цей файл використовувати (найкраще – для читання і записування):

```
fd1 = open ("myfile", O_RDWR | O_CREAT, 0644);
```

2. Якщо це новий файл, задати його довжину за допомогою `lseek()` і `write()`:

```
lseek(fd1, len, SEEK_SET); write(fd1, "", 1);
```

3. Відобразити файл у пам'ять:

```
int *fmap = (int*)mmap(0, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);
```

4. Закрити дескриптор файла:

```
close(fd1);
```

5. Організувати доступ до файла через відображувану пам'ять:

```
fmap[0] = 100; fmap[1] = fmap[0] * 2;
```

6. Якщо відображення більше не потрібне – скасувати його:

```
munmap(fmap, len);
```

Під час роботи із відображуваною пам'яттю застосування може отримувати сигнали `SIGSEGV` (у разі спроби записати у пам'ять, відкриту тільки для читання) і `SIGBUS` (у разі спроби доступу до ділянки за межами відображення, наприклад, за кінцем файла).

Міжпроцесова взаємодія на основі відображуваної пам'яті POSIX

Кілька процесів можуть відобразити у свої адресні простори той самий файл (якщо задано режим відображення `MAP_SHARED`). Зміни у відображеній пам'яті, зроблені одним із цих процесів, будуть негайно видимі в усіх інших. Таким чином реалізують розподілювану пам'ять.

Наведемо приклад обміну даними між процесами за допомогою відображуваної пам'яті. Одне із застосувань буде сервером і очікуватиме змін, зроблених клієнтом.

```
int *fmap; int fd1;
fd1 = open("myfile", O_RDWR | O_CREAT, 0644);
lseek(fd1, sizeof(int), SEEK_SET);  write(fd1, "", 1);
// відображення
```

```
fmap = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);
close(fd1);
fmap[0] = 1;  // задання початкового значення
// читання значення в циклі (можливо, зміненого іншим процесом)
for (; ;) {
    printf("%d\n", fmap[0]);
    sleep(1);
}
```

Клієнт буде подвоювати значення у відображуваній пам'яті.

```
int *fmap; int fd1;
fd1 = open("./myfile", O_RDWR, 0644);
// відображення
fmap = (int*)mmap(0, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);
close(fd1);
// зміна значення
fmap[0] *= 2;
```

У разі використання відображуваної пам'яті для міжпроцесової взаємодії потрібно забезпечувати синхронізацію доступу до неї. Для цього можна використати засоби міжпроцесової синхронізації, наприклад файлові блокування.

```
// запровадити файлове блокування, див. 11.6.1
fmap[0] *= 2;
// зняти файлове блокування
```

Особливості інтерфейсу відображуваної пам'яті Win32 API

Як параметр у функцію відображення файла в пам'ять (у Win32 API її називають `MapViewOfFile()`) передають не дескриптор відкритого файла, як у `mmap()`, а дескриптор спеціального *об'єкта відображення*, що створюється заздалегідь за допомогою функції `CreateFileMapping()`. Під час створення об'єкта відображення вказують дескриптор відкритого файла; далі цей об'єкт можна використовувати для відображення кількох регіонів одного файла.

Розглянемо особливості функції `CreateFileMapping()`:

```
HANDLE CreateFileMapping(HANDLE fh, LPSECURITY_ATTRIBUTES psa,
   DWORD prot, DWORD size_high, DWORD size_low, LPCTSTR mapname);
```

де: `hf` — дескриптор файла, відкритого для читання (а можливо, і для записування);

`prot` — режим доступу до пам'яті (`PAGE_READWRITE` — розподілений доступ для читання і записування, аналогічний до `MAP_SHARED`, `PAGE_WRITECOPY` — копіювання під час записування, аналогічне до `MAP_PRIVATE`);

`size_low`, `size_high` — розмір файла (може дорівнювати нулю, у цьому разі розмір відображення буде рівним поточному розміру файла, `size_high` використовують, коли файл за розміром більший за 232 байти);

`mapname` – ім'я відображення.

`CreateFileMapping()` повертає дескриптор об'єкта відображення.

Якщо файл займає менше місця, ніж задано за допомогою параметрів `size_low` і `size_high`, його автоматично розширюють до потрібного розміру без необхідності виклику аналогів `lseek()` і `write()`.

Після створення об'єкта відображення необхідно на його основі задати відображення у пам'ять за допомогою MapViewOfFile():

```
PVOID MapViewOfFile (HANDLE fmap, DWORD prot, DWORD off_high,
  DWORD off_low, SIZE_T len);
```

де: fmap — дескриптор об'єкта відображення;

prot — режим відображення (FILE_MAP_WRITE — доступ для записування, потребує задання PAGE_READWRITE для об'єкта відображення, FILE_MAP_COPY — копіювання під час записування, потребує задання PAGE_WRITECOPY);

off_low, off_high — зсув відображуваного регіону від початку файла (має бути кратним 64 Кбайт);

len — довжина відображуваного регіону у байтах (якщо len=0, відображають ділянку до кінця файла).

Ця функція повертає покажчик на початок відображеної ділянки пам'яті. Аналогом munmap() є функція UnmapViewOfFile():

```
BOOL UnmapViewOfFile(PVOID start);
```

де start — початкова адреса ділянки відображення.

Після використання дескриптор файла і дескриптор об'єкта відображення потрібно закрити за допомогою CloseHandle(). Дескриптор файла можна закрити і до виклику MapViewOfFile().

Для роботи з відображуваною пам'яттю у Win32 необхідно.

1. Відкрити файл, звичайно для читання і для записування:

```
HANDLE fh = CreateFile ("myfile.txt", GENERIC_READ|GENERIC_WRITE, 0,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
```

2. Створити об'єкт відображення для файлу:

```
HANDLE mh = CreateFileMapping (fh, 0, PAGE_READWRITE, 0, len, NULL);
```

3. Закрити файл:

```
CloseHandle(fh);
```

4. Відобразити файл або його фрагмент у пам'ять:

```
int *fmap = (int*)MapViewOfFile(mh, FILE_MAP_WRITE, 0, 0, len);
```

5. Організувати доступ до файла через відображувану пам'ять:

```
fmap[0] = 100; fmap[1] = fmap[0] * 2;
```

6. Якщо відображення більше не потрібне — закрити відповідний дескриптор:

```
CloseHandle(fh);
```

Реалізація розподілюваної пам'яті у Win32 API

Ґрунтуючись на підтримці відображуваної пам'яті у Win32 API, легко реалізувати розподіл пам'яті між процесами. Це зумовлено низкою можливостей, пов'язаних із наявністю окремих об'єктів відображення.

- Можна задавати об'єкти відображення, які використовують не конкретний дисковий файл, а файл підкачування; для цього достатньо передати у функцію CreateFileMapping() замість дескриптора файла значення INVALID_HANDLE_VALUE (–1).

Це позбавляє програміста необхідності створювати файл тільки для обміну даними з іншим процесом (як потрібно у POSIX).

- Об'єктам відображення під час їхнього створення можна давати імена, після чого вони можуть використовуватися кількома процесами: перший створює об'єкт і задає його ім'я (як параметр CreateFileMapping()), а інші відкривають його за допомогою функції OpenFileMapping(), передавши як параметр те саме ім'я.

Функцію OpenFileMapping() використовують для отримання доступу до наявного об'єкта відображення:

```
HANDLE OpenFileMapping(DWORD prot, BOOL isinherit, LPCSTR name);
```

де: prot — аналогічний відповідному параметру MapViewOfFile();

name — ім'я об'єкта відображення.

Приклад обміну даними між клієнтом і сервером, аналогічний до прикладу для відображуваної пам'яті POSIX, наведено нижче.

```
// сервер
HANDLE mh = CreateFileMapping (INVALID_HANDLE_VALUE, 0, PAGE_READWRITE,
0, sizeof(int), "mymap");
int *fmap = (int*)MapViewOfFile (mh, FILE_MAP_WRITE, 0, 0, sizeof(int));
fmap[0] = 1;
for (; ;) {
    printf("%d\n", fmap[0]);
    Sleep(1000);
}
UnmapViewOfFile (fmap);
CloseHandle (mh);

// клієнт
HANDLE mh = OpenFileMapping (FILE_MAP_WRITE, 0, "mymap");
int *fmap = (int*)MapViewOfFile (mh, FILE_MAP_WRITE, 0, 0, sizeof(int));
fmap[0] *= 2;
UnmapViewOfFile (fmap);
CloseHandle (mh);
```

Особливості реалізації відображуваної пам'яті

Практична реалізація відображуваної пам'яті звичайно спирається на функції менеджера віртуальної пам'яті. У разі відображення файла у пам'ять створюють зв'язок між файлом і пам'яттю, аналогічний до того, що задають між пам'яттю і файлом підкачування під час завантаження сторінок на вимогу (див. розділ 9). За першої спроби доступу до такої пам'яті відбувається сторінкове переривання, оброблювач якого завантажує сторінки із файла безпосередньо у пам'ять (в адресний простір процесу). Під час записування відповідну сторінку пам'яті позначають як модифіковану, подальша робота з нею відбувається аналогічно до роботи зі звичайною модифікованою сторінкою у разі підкачування (наприклад, якщо відбувається заміщення, її автоматично збережуть у файлі).

Переваги й недоліки відображуваної пам'яті

Наведемо переваги використання файлів, що відображаються у пам'ять.

- Робота із такими файлами зводиться до прямого звертання до пам'яті через покажчики, ніби файл був частиною адресного простору процесу. Під час роботи

зі звичайними файлами для виконання одних і тих самих дій потрібно відкривати файл, переміщувати покажчик поточної позиції, виконувати читання і записування та закривати файл.

- Робота із відображуваними файлами може бути реалізована ефективніше. Це пов'язано з тим, що:
  - для роботи з ними достатньо виконати один системний виклик (`mmap()` або `MapViewOfFile()`), а далі працювати із ділянкою пам'яті в адресному просторі процесу; водночас для відкриття файла, читання, записування тощо потрібно виконувати окремі системні виклики;
  - не потрібно копіювати дані між системною пам'яттю і буфером режиму користувача, що необхідно для звичайних операцій читання і записування файла — у разі використання цієї технології файл безпосередньо відображається на адресний простір процесу.
- За допомогою відображуваних файлів можна легко реалізувати розподіл пам'яті між процесами, якщо відобразити один і той самий файл на адресний простір кількох із них.

Головним недоліком цієї технології є те, що відображуваний файл не може бути розширений позиціюванням покажчика поточної позиції за його кінець і виконанням операції записування (через те, що підсистема віртуальної пам'яті у цьому випадку не може визначити точну довжину файла).

Приклади використання відображуваної пам'яті

Назвемо деякі підходи до використання відображуваної пам'яті на рівні операційної системи.

- Відображення файла у пам'ять може використати для завантаження виконуваних файлів у пам'ять (такі файли відображаються в адресний простір процесу і є простором підтримки для сторінок коду). Про приклад використання такої технології у Windows XP ішлося у розділі 9.9.1. За таким самим принципом завантажують виконувані файли в Linux.
- На основі відображуваної пам'яті можна також реалізувати кешування диска; про такий підхід ітиметься у розділі 12.
- За допомогою цієї технології можна виділяти пам'ять процесу. Для цього в його адресний простір відображають спеціальний файл, у разі доступу до якого завжди повертають нулі. В UNIX його називають `/dev/zero`.

```
int zero = open("/dev/zero", O_RDWR, 0644);
char *area = (char*)mmap(0, getpagesize(), PROT_READ | PROT_WRITE,
                         MAP_SHARED, zero, 0);
// area вказує на сторінку пам'яті, заповнену нулями
```

- Можна забезпечити автоматичне збереження значень складних структур даних між викликами програми (такі структури створюють у ділянці пам'яті, що відповідає відображеному файлу; під час наступних викликів цей файл відображають у ту саму ділянку пам'яті знову, і всі структури поновлюються в цій пам'яті повністю).

11.6.3. Поіменовані канали

Поіменовані канали POSIX

Поіменовані канали POSIX [24, 37] є однобічним засобом передавання даних. Це означає, що якщо один процес записує дані в канал, то інший із цього каналу може тільки читати, причому читання даних відбувається в порядку їхнього записування за принципом FIFO.

Поіменовані канали відображені спеціальними FIFO-файлами у файловій системі UNIX. За іменем такого файла до поіменованого каналу може підключитися будь-який процес у системі, у якого є права читання з цього файла. Після цього всі дані, передані в канал, надходитимуть цьому процесу, поки канал не буде закрито.

Для створення FIFO-файла у POSIX передбачена функція `mkfifo()`:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t mode);
```

Першим параметром `mkfifo()` приймає повний шлях до створюваного файла, другим — режим (аналогічно до першого і третього параметрів `open()`). Ось приклад створення поіменованого каналу:

```
mkfifo ("./myfifo", 0644);
```

Після того як канал (FIFO-файл) створено, будь-який процес може підключитися до нього. Для цього він має спочатку створити з'єднання із каналом за допомогою виклику `open()`. При цьому процес буде заблокований для очікування даних від іншого процесу (у разі читання із каналу очікування триватиме доти, поки інший процес не запише у канал, у разі записування — доки не буде виконане читання). Після закінчення очікування процес може записувати дані в канал за допомогою `write()` або зчитувати їх із каналу за допомогою `read()`:

```
int fd1 = open("./myfifo", O_RDONLY);   // очікування даних
read(fd1, read_buf, sizeof(read_buf));  // дані надійшли – читаємо їх
```

Розглянемо найпростіший підхід до обміну даними через поіменований канал (коли клієнти посилають на сервер повідомлення).

Для цього на сервері потрібно виконати такий код:

```
int fd1, bytes_read; char read_buf[100];
mkfifo ("/dir/myfifo", 0644);   // створити файл каналу
for (; ;) {  // нескінченний цикл (вихід, наприклад, за сигналом)
    fd1 = open("/dir/myfifo", O_RDONLY);
    bytes_read = read(fd1, read_buf, sizeof(read_buf));
    if (bytes_read > 0)  {
        read_buf[bytes_read] = '\0';
        printf("сервер: отримано повідомлення: %s\n", read_buf);
    }
    close(fd1);   // закрити канал
}
```

У разі закриття сервера (наприклад, за сигналом) необхідно закрити канал і вилучити файл каналу за допомогою `unlink()`. Наведемо код клієнта:

```
fd1 = open("/dir/myfifo", O_WRONLY);     // відкрити канал для записування
write(fd1, "hello", sizeof("hello"));    // записати в нього дані
close(fd1);   // закрити канал
```

Поіменовані канали Win32 API

На відміну від каналів POSIX, поіменовані канали Win32 [9, 31] реалізують двобічний обмін повідомленнями і не використовують прямо файли на файловій системі. Однак їхнє використання ґрунтується на схожих принципах.

Імена таких каналів створюють за такою схемою: \\ім'я_машини\pipe\ім'я_каналу. Для доступу до каналу на локальному комп'ютері замість імені машини можна використовувати символ «.»: \\.\pipe\ім'я_каналу. Такі імена називають UNC-іменами (Universal Naming Convention – універсальна угода з імен); про них докладно йтиметься в розділі 16.

Для створення екземпляра каналу на сервері використовують функцію CreateNamedPipe():

```
HANDLE pipe = CreateNamedPipe(LPCTSTR pname, DWORD openmode,
   DWORD pipemode, DWORD max_conn, DWORD obufsize, DWORD ibufsize,
   DWORD timeout, LPSECURITY_ATTRIBUTES psa);
```

де: pname – ім'я каналу на локальному комп'ютері (на віддалених комп'ютерах створювати канали не можна);

openmode – режим відкриття каналу (наприклад, вмикання прапорця PIPE_ACCESS_DUPLEX означає двобічний зв'язок);

pipemode – режим обміну даними із каналом (наприклад, для побайтового читання із каналу потрібно увімкнути прапорець PIPE_READMODE_BYTE, для побайтового записування – PIPE_TYPE_BYTE);

max_conn – максимальна кількість клієнтських з'єднань із цим каналом (необмежену кількість задають як PIPE_UNLIMITED_INSTANCES);

timeout – максимальний час очікування клієнтом доступу до цього каналу (необмежене очікування задають як NMPWAIT_WAIT_FOREVER).

CreateNamedPipe() повертає дескриптор створеного каналу:

```
HANDLE ph = CreateNamedPipe("\\\\.\\pipe\\mypipe", PIPE_ACCESS_DUPLEX,
PIPE_READMODE_BYTE, 5, 0, 0, NMPWAIT_WAIT_FOREVER, NULL);
```

Для очікування клієнтського з'єднання необхідно виконати функцію ConnectNamedPipe():

```
ConnectNamedPipe(ph, NULL);
```

Як тільки клієнт підключиться до цього каналу, очікування буде завершене, і можна читати з каналу або записувати в канал за допомогою ReadFile() і WriteFile(). Після завершення обміну даними сервер закриває з'єднання, викликавши функцію DisconnectNamedPipe().

Після завершення роботи із каналом (наприклад, у разі виходу із програми) його дескриптор потрібно закрити за допомогою CloseHandle().

Наведемо код сервера, аналогічний за функціональністю до прикладу для поіменованих каналів POSIX.

```
DWORD bytes_read; char buf[100];
HANDLE ph = CreateNamedPipe("\\\\.\\pipe\\mypipe", PIPE_ACCESS_DUPLEX,
PIPE_READMODE_BYTE, 1, 0, 0, NMPWAIT_WAIT_FOREVER, NULL);
for (; ;) {
    ConnectNamedPipe (ph, NULL);  // почати прослуховування каналу
```

```
    // прочитати дані з каналу (від клієнта)
    ReadFile(ph, buf, sizeof(buf), &bytes_read, 0);
    DisconnectNamedPipe (ph);     // закрити з'єднання
}
CloseHandle(ph);  // закрити дескриптор каналу
```

При розробці коду клієнта, необхідно:

- викликати функцію WaitNamedPipe(), що очікуватиме виконання сервером ConnectNamedPipe();
- після того, як очікування завершене, встановити з'єднання із каналом за допомогою CreateFile();
- виконати обмін даними за допомогою ReadFile() і WriteFile(), використовуючи дескриптор з'єднання, повернений CreateFile();
- закрити дескриптор каналу за допомогою CloseHandle():

```
DWORD bytes_written;
WaitNamedPipe("\\\\.\\pipe\\mypipe", NMPWAIT_WAIT_FOREVER);
HANDLE pipeh = CreateFile("\\\\.\\pipe\\mypipe", GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
WriteFile(pipeh, "hello", sizeof("hello"), &bytes_written, 0);
CloseHandle(pipeh);
```

Зауважимо, що у функції WaitNamedPipe() і CreateFile() можна передавати імена каналів, розміщених на віддалених комп'ютерах.

Висновки

- Для універсального доступу до інформації із прикладних програм ОС надають інтерфейс файлових систем. Файлову систему можна розглядати на двох рівнях: логічному і фізичному. На логічному рівні файлові системи абстрагують доступ до дискового простору або до інших пристроїв у вигляді концепції файлів, розташованих у каталогах. На фізичному рівні, вони реалізують набір структур даних, що зберігаються на пристроях, які забезпечують логічне відображення.
- Файлом називають набір даних на файловій системі, до якого можна звертатися за іменем. Файли є основним засобом реалізації зберігання даних в енергонезалежній пам'яті і забезпечують їхнє спільне використання різними процесами. Сучасні ОС звичайно не виділяють структуру файла, зображуючи його як послідовність байтів. Винятками є спеціальні файли. Операції доступу до файлів (відкриття, закриття, читання, записування тощо) — це універсальний інтерфейс доступу до даних в операційних системах. Важливим підходом до використання файлів у сучасних ОС стала реалізація на їхній основі відображуваної та розподілюваної пам'яті.
- Сучасні файлові системи мають структуру дерева або графа каталогів. Підходи до організації структури каталогів різні для різних ОС: деякі системи реалізують єдине дерево або граф каталогів, куди можуть підключатися (монтуватися) окремі файлові системи, розміщені на розділах диска; інші системи позначають окремі розділи диска буквами алфавіту і працюють із відповідними

файловими системами окремо. Дерево каталогів стає графом за наявності зв'язків. Зв'язки можуть бути жорсткими (альтернативне ім'я для файла) і символічними (файл, що містить посилання на інший файл або каталог).

Контрольні запитання та завдання

1. Перелічіть переваги і недоліки відображення файла у вигляді неструктурованого потоку байтів.
2. Які проблеми виникають, якщо дозволено одночасне монтування файлової системи в кілька точок монтування?
3. У деякій ОС файли автоматично відкриваються під час першого звертання до них і закриваються — у разі завершення процесу. Опишіть переваги і недоліки цього підходу порівняно з традиційним, коли файли відкривають і закривають явно.
4. Яка помилка може виникати під час виконання наступного коду? У яких умовах вона проявляється?

```
void fileCopy(int fsrc, int fdest){
    char buf[100];
    while(read(fsrc,buf,100) > 0)
       write(fdest,buf,100);
}
```

5. Напишіть функцію, внаслідок виконання якої вміст масиву цілих чисел (типу int) записується у файл, починаючи з позиції, що відповідає його середині. Файл завдяки цьому збільшується: елементи його другої половини зміщуються до кінця файла, розташовуючись після даних масиву. Параметрами функції є дескриптор файла, покажчик на початок масиву і кількість елементів у масиві.
6. Наведіть приклади взаємних блокувань, що можуть виникати у разі використання файлових блокувань.
7. Реалізуйте набір функцій обробки файлів, що містять записи фіксованої довжини. У набір повинні входити функції створення, відкриття і закриття файла, читання і відновлення значення запису за його номером, перегляду всіх записів. У разі відновлення файла всі спроби інших процесів оновити той самий запис повинні бути блоковані. Якщо під час спроби відновлення запис із таким номером не було знайдено, має бути створено новий запис.
8. У чому полягають основні відмінності реалізації відображуваної пам'яті в Linux і Windows XP? Який із підходів видається більш гнучким?
9. Які проблеми виникають у разі спроби розв'язати завдання 5 і 7 цього розділу з використанням файлів, відображуваних у пам'ять?
10. Реалізуйте операцію копіювання файлів у Linux і Windows XP на базі інтерфейсу файлів, відображуваних у пам'ять.
11. Розробіть систему обміну даними про поточну температуру повітря для Linux і Windows XP з використанням відображуваної пам'яті. Інформація про температуру є цілим числом. Є N процесів-клієнтів, які повинні зчитувати і ві-

дображати цю інформацію, і один процес-менеджер, якому дозволено її змінювати. Зміни, зроблені менеджером, мають відображати всі клієнти.

12. Розробіть просту клієнт-серверну систему для Linux і Windows XP з використанням поіменованих каналів. Клієнт приймає від користувача шлях файла та передає його на сервер. Сервер повинен знаходити на диску відповідний файл і направляти його вміст клієнту, котрий після отримання цих даних має відобразити їх. Якщо файл не знайдено, сервер повертає рядок з повідомленням про помилку. У разі одержання рядка «exit» сервер повинен завершити свою роботу.

Розділ 12

Фізична організація і характеристики файлових систем

- Організація розділів на жорсткому диску
- Фізичне розміщення файлів і каталогів
- Організація дискового кеша
- Дискове планування
- Резервне копіювання
- Журнальні файлові системи

У цьому розділі розглянемо основні підходи до розміщення інформації на файлових системах, а також питання забезпечення їхньої продуктивності та надійності.

12.1. Базові відомості про дискові пристрої

У цьому розділі зупинимося на особливостях дискових пристроїв, що впливають на реалізацію доступу до таких пристроїв у ОС. Нас цікавитимуть жорсткі диски, інформація про особливості реалізації та використання інших типів дискових накопичувачів (компакт-дисків, гнучких дисків тощо) можна знайти, наприклад, у [44].

12.1.1. Принцип дії жорсткого диска

Накопичувачі на жорстких магнітних дисках (НЖМД) (рис. 12.1, далі – диски) складаються з набору дискових пластин (platters), які покриті магнітним матеріалом і обертаються двигуном із високою швидкістю. Кожній пластині відповідають дві головки (heads), одна зчитує інформацію зверху, інша – знизу. Головки прикріплені до спеціального дискового маніпулятора (disk arm). Маніпулятор може переміщатися по радіусу диска – від центра до зовнішнього краю і назад, таким чином відбувається позиціювання головок.

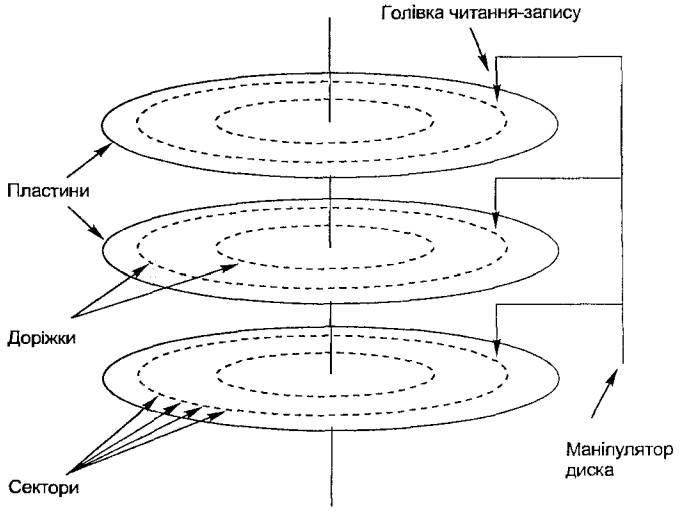

Рис. 12.1. Базовий устрій накопичувача на жорстких магнітних дисках

Головки зчитують інформацію із *доріжок* (tracks), які мають вигляд концентричних кіл. Мінімальна кількість доріжок на поверхні пластини в сучасних дисках – 700, максимальна – більше 20 000. Сукупність усіх доріжок одного радіуса на всіх поверхнях пластин називають *циліндром.*

Кожну доріжку під час низькорівневого форматування розбивають на *сектори* (sectors), обсяг даних сектора для більшості архітектур становить 512 байт (він обов'язково має дорівнювати степеню числа 2). Кількість секторів для всіх доріжок однакова (у діапазоні від 16 до 1600).

12.1.2. Ефективність операцій доступу до диска

Основними характеристиками доступу до диска є:

- *час пошуку* (seek time) – час переміщення маніпулятора для позиціювання головки на потрібній доріжці (у середньому становить від 10 до 20 мс);
- *ротаційна затримка* (rotational delay) – час очікування, поки пластина повернеться так, що потрібний сектор опиниться під доріжкою (у середньому становить 8 мс);
- *пропускна здатність передавання даних* (transfer bandwidth) – обсяг даних, що передаються від пристрою в пам'ять за одиницю часу; для сучасних дисків ця характеристика порівняння із пропускною здатністю оперативної пам'яті (200 Мбайт/с), час передавання одного сектора вимірюють у наносекундах.

Час, необхідний для читання сектора, одержують додаванням часу пошуку, ротаційної затримки і часу передавання (при цьому час передавання можна вважати дуже малим). Очевидно, що час читання одного сектора практично не відрізняється від часу читання кількох розташованих поряд секторів, а час читання цілої доріжки за одну операцію буде менший, ніж час читання одного сектора через відсутність ротаційної затримки.

Швидкість доступу до диска, порівняно із доступом до пам'яті, надзвичайно мала, зараз диски є основним «вузьким місцем» з погляду продуктивності комп'ютерної системи. При цьому реальне поліпшення останніми роками наявне лише для пропускної здатності передавання даних. Час пошуку і ротаційна затримка майже не змінюються, тому що вони пов'язані з керуванням механічними пристроями (дисковим маніпулятором і двигуном, що обертає пластини) і обмежені їхніми фізичними характеристиками.

У результаті час читання великих обсягів неперервних даних усе менше відрізняється від часу читання малих. Як наслідок, першорядне значення для розробників файлових систем набуває розв'язання двох задач:

- організації даних таким чином, щоб ті з них, які будуть потрібні одночасно, перебували на диску поруч (і їх можна було зчитати за одну операцію);
- підвищення якості кешування даних (оскільки пам'яті стає все більше, зростає ймовірність того, що всі потрібні дані міститимуться в кеші, і доступ до диска стане взагалі не потрібний).

12.2. Розміщення інформації у файлових системах

Файлова система звичайно будує базове відображення даних поверх того, яке їй надають драйвери дискових пристроїв. Насамперед, ОС розподіляє дисковий простір не секторами, а спеціальними одиницями розміщення – *кластерами* (clusters) або *дисковими блоками* (disk blocks, термін «дисковий блок» більш розповсюджений в UNIX-системах). Визначення розміру кластера і розміщення інформації, необхідної для функціонування файлової системи, відбувається під час високорівневого форматування розділу. Саме таке форматування створює файлову систему в розділі.

Розмір кластера визначає особливості розподілу дискового простору в системі. Використання кластерів великого розміру може спричинити значну внутрішню фрагментацію через файли, які за розміром менші, ніж кластер.

Деякі застосування (насамперед, сервери баз даних) можуть реалізовувати свою власну фізичну організацію даних на диску. Для них файлова система може виявитися зайвим рівнем доступу, що тільки сповільнюватиме роботу. Багато ОС надають таким застосуванням можливість працювати із розділами, поданими у вигляді простого набору дискових секторів, який не містить структур даних файлової системи. Про такі розділи кажуть, що вони містять неорганізовану файлову систему (raw file system). Для них не виконують операцію високорівневого форматування.

12.2.1. Фізична організація розділів на диску

Перед тим як перейти до розгляду особливостей фізичної організації файлової системи в рамках розділу, коротко ознайомимося з організацією розділів на диску.

Початковий (нульовий) сектор диска називають *головним завантажувальним записом* (Master Boot Record, MBR). Наприкінці цього запису міститься *таблиця розділів* цього диска, де для кожного розділу зберігається початкова і кінцева адреси.

Один із розділів диска може бути позначений як *завантажувальний* (bootable) або *активний* (active). Після завантаження комп'ютера апаратне забезпечення звертається до MBR одного з дисків, визначає з його таблиці розділів завантажувальний розділ і намагається знайти в першому кластері цього розділу спеціальну невелику програму — *завантажувач ОС* (OS boot loader). Саме завантажувач ОС відповідає за пошук на диску і початкове завантаження у пам'ять ядра операційної системи. Докладніше про роботу завантажувача ОС ітиметься в розділі 19.

Усередині розділу розташовані структури даних файлової системи.

12.2.2. Основні вимоги до фізичної організації файлових систем

З погляду користувача файл із заданим іменем (далі вважатимемо, що інформацію про шлях включено в ім'я) — це неструктурована послідовність байтів, а з погляду фізичної структури файлової системи, файл — це набір дискових блоків, що містять його дані. Завдання файлової системи полягає у забезпеченні перетворення сукупності імені файла і логічного зсуву в ньому на фізичну адресу всередині відповідного дискового блоку.

Необхідність такого перетворення визначає основне завдання файлової системи — відстежувати розміщення вмісту файлів на диску. Інформація про розміщення даних файла на диску зберігається у структурі даних, що називають *заголовком файла*. Такі заголовки звичайно зберігають на диску разом із файлами. Під час розробки структури даних для такого заголовка потрібно враховувати, що більшість файлів мають малий розмір, а основну частину дискового простору розподіляють, навпаки, під файли великого розміру, із якими переважно і виконують операції введення-виведення.

Оскільки продуктивність файлової системи залежить від кількості операцій доступу до диска, важливо максимально її обмежити. Кілька сотень таких операцій можуть додатково зайняти кілька секунд часу. На практиці слід враховувати, що всі імена файлів (і самі файли) каталогу і всі блоки у файлі зазвичай використовують разом, послідовно.

Принципи, що лежать в основі фізичної організації файлової системи, визначають різні способи розміщення файлів на диску. Крім обліку розміщення даних, фізичне розміщення потребує також обліку вільних кластерів. Різні варіанти організації такого обліку наведені у розділі 12.2.7.

12.2.3. Неперервне розміщення файлів

Найпростіший підхід до фізичної організації файлових систем — це неперервне розміщення файлів. При цьому кожному файлові відповідає набір неперервно розташованих кластерів на диску (рис. 12.2). Для кожного файла мають зберігатися адреса початкового кластера і розмір файла.

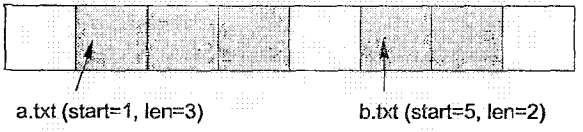

Рис. 12.2. Неперервне розміщення файлів

Зазначимо, що розподіл дискового простору в цьому разі подібний до динамічного розподілу пам'яті. Для пошуку вільного блоку на диску можна використати алгоритми першого підходящого або найкращого підходящого блоку.

Неперервне розміщення файлів вирізняється простотою в реалізації та ефективністю (наприклад, весь файл може бути зчитаний за одну операцію), але має істотні недоліки.

- Під час створення файла користувач має заздалегідь задати його максимальну довжину і виділити весь простір на диску за один раз. Збільшувати розміри файлів під час роботи не можна. У багатьох ситуаціях це абсолютно неприйнятно (наприклад, неможливо вимагати від користувача текстового редактора щоб він вказував остаточну довжину файла перед його редагуванням).
- Вилучення файлів згодом може спричинити велику зовнішню фрагментацію дискового простору з тих самих причин, що й за динамічного розподілу пам'яті.

У сучасних ОС для організації даних на жорстких дисках неперервне розміщення майже не використовують, проте його застосовують у таких файлових системах, де можна заздалегідь передбачити, якого розміру буде файл. Прикладом є файлові системи для компакт-дисків. Вони мають кілька властивостей, що роблять неперервне розміщення файлів найкращим рішенням:

- записування такої файлової системи здійснюють повністю за один раз, під час записування для кожного файла заздалегідь відомий його розмір;
- доступ до файлових систем на компакт-диску здійснюють лише для читання, файли в них ніколи не розширюють і не вилучають, тому відсутні причини появи зовнішньої фрагментації.

12.2.4. Розміщення файлів зв'язними списками

Прості зв'язні списки

Іншим підходом є організація кластерів, що належать файлу, у зв'язний список. Кожен кластер файла містить інформацію про те, де перебуває наступний кластер цього файла (наприклад, його номер). Найпростіший приклад такого розміщення бачимо на рис. 12.3. Заголовок файла в цьому разі має містити посилання на його перший кластер, вільні кластери можуть бути організовані в аналогічний список.

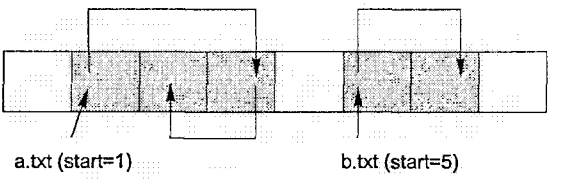

Рис. 12.3. Найпростіший приклад зв'язаного розміщення файлів

Розміщення файлів з використанням зв'язних списків надає такі переваги:

- відсутність зовнішньої фрагментації (є тільки невелика внутрішня фрагментація, пов'язана з тим, що розмір файла може не ділитися націло на розмір кластера);
- мінімум інформації, яка потрібна для зберігання у заголовку файла (тільки посилання на перший кластер);

- можливість динамічної зміни розміру файла;
- простота реалізації керування вільними блоками, яке принципово не відрізняється від керування розміщенням файлів.

Цей підхід, однак, не позбавлений і серйозних недоліків:

- відсутність ефективної реалізації випадкового доступу до файла: для того щоб одержати доступ до кластера з номером n, потрібно прочитати всі кластери файла з номерами від 1 до n-1;
- зниження продуктивності тих застосувань, які зчитують дані блоками, за розміром рівними степеню числа 2 (а таких застосувань досить багато): частина будь-якого кластера повинна містити номер наступного, тому корисна інформація в кластері займає обсяг, не кратний його розміру (цей обсяг навіть не є степенем числа 2);
- можливість втрати інформації у послідовності кластерів: якщо внаслідок збою буде втрачено кластер на початку файла, вся інформація в кластерах, що йдуть за ним, також буде втрачена.

Є модифікації цієї схеми, які зберегли своє значення дотепер, найважливішою з них є використання таблиці розміщення файлів.

Зв'язні списки з таблицею розміщення файлів

Цей підхід (рис. 12.4) полягає в тому, що всі посилання, які формують списки кластерів файла, зберігаються в окремій ділянці файлової системи фіксованого розміру, формуючи *таблицю розміщення файлів* (File Allocation Table, FAT). Елемент такої таблиці відповідає кластеру на диску і може містити:

- номер наступного кластера, якщо цей кластер належить файлу і не є його останнім кластером;
- індикатор кінця файла, якщо цей кластер є останнім кластером файла;
- індикатор, який показує, що цей кластер вільний.

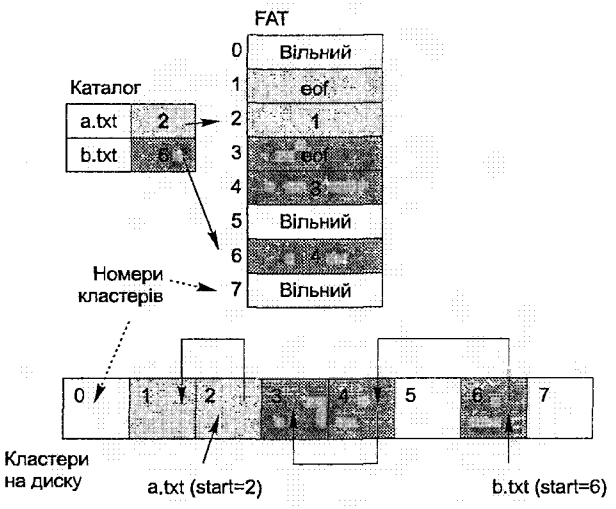

Рис. 12.4. Використання таблиці розміщення файлів

Для організації файла достатньо помістити у відповідний йому елемент каталога номер першого кластера файла. За необхідності прочитати файл система знаходить за цим номером кластера відповідний елемент FAT, зчитує із нього інформацію про наступний кластер і т. д. Цей процес триває доти, поки не трапиться індикатор кінця файла.

Використання цього підходу дає змогу підвищити ефективність і надійність розміщення файлів зв'язними списками. Це досягається завдяки тому, що розміри FAT дозволяють кешувати її в пам'яті. Через це доступ до диска під час відстеження посилань заміняють звертаннями до оперативної пам'яті. Зазначимо, що навіть якщо таке кешування не реалізоване, випадковий доступ до файла не призводитиме до читання всіх попередніх його кластерів — зчитані будуть тільки попередні елементи FAT.

Крім того, спрощується захист від збоїв. Для цього, наприклад, можна зберігати на диску додаткову копію FAT, що автоматично синхронізуватиметься з основною. У разі ушкодження однієї з копій інформація може бути відновлена з іншої.

І нарешті, службову інформацію більше не зберігають безпосередньо у кластерах файла, вивільняючи в них місце для даних. Тепер обсяг корисних даних всередині кластера майже завжди (за винятком, можливо, останнього кластера файла) дорівнюватиме степеню числа 2.

Однак, у разі такого способу розміщення файлів для розділів великого розміру обсяг FAT може стати доволі великим і її кешування може потребувати значних витрат пам'яті. Скоротити розмір таблиці можна, збільшивши розмір кластера, але це, в свою чергу, призводить до збільшення внутрішньої фрагментації для малих файлів (менших за розмір кластера).

Також руйнування обох копій FAT (внаслідок апаратного збою або дії програми-зловмисника, наприклад, комп'ютерного вірусу) робить відновлення даних дуже складною задачею, яку не завжди можна розв'язати.

Про реалізацію цього підходу йтиметься в розділі 13 під час знайомства із файловими системами лінії FAT.

12.2.5. Індексоване розміщення файлів

Базовою ідеєю ще одного підходу до розміщення файлів є перелік адрес всіх кластерів файла в його заголовку. Такий заголовок файла дістав назву індексного дескриптора, або і-вузла (inode), а сам підхід — індексованого розміщення файлів.

За індексованого розміщення із кожним файлом пов'язують його індексний дескриптор. Він містить масив із адресами (або номерами) усіх кластерів цього файла, при цьому *n-й* елемент масиву відповідає *n-му* кластеру. Індексні дескриптори зберігають окремо від даних файла, для цього звичайно виділяють на початку розділу спеціальну ділянку індексних дескрипторів. В елементі каталогу розміщують номер індексного дескриптора відповідного файла (рис. 12.5).

Під час створення файла на диску розміщують його індексний дескриптор, у якому всі покажчики на кластери спочатку є порожніми. Під час першого записування в *n-й* кластер файла менеджер вільного простору виділяє вільний кластер і його номер або адресу заносять у відповідний елемент масиву.

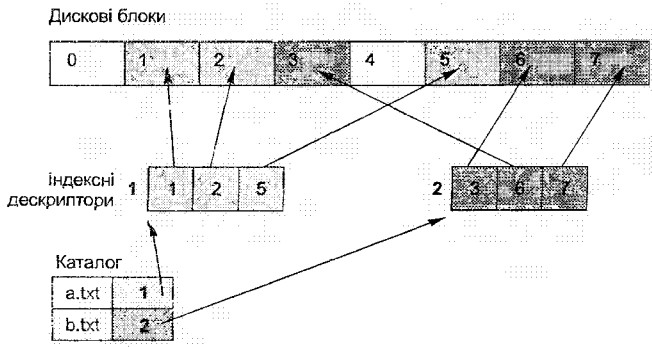

Рис. 12.5. Індексоване розміщення файлів

Цей підхід стійкий до зовнішньої фрагментації й ефективно підтримує як послідовний, так і випадковий доступ (інформація про всі кластери зберігається компактно і може бути зчитана за одну операцію). Для підвищення ефективності індексний дескриптор повністю завантажують у пам'ять, коли процес починає працювати з файлом, і залишають у пам'яті доти, поки ця робота триває.

Структура індексних дескрипторів

Основною проблемою є підбір розміру і задання оптимальної структури індексного дескриптора, оскільки:

- з одного боку, зменшення розміру дескриптора може значно зекономити дисковий простір і пам'ять (дескрипторів потрібно створювати значну кількість – по одному на кожний файл, разом вони можуть займати досить багато місця на диску; крім того, для кожного відкритого файла дескриптор буде розташовано в оперативній пам'яті).
- з іншого боку, дескриптора надто малого розміру може не вистачити для розміщення інформації про всі кластери великого файла.

Одне з компромісних розв'язань цієї задачі, яке застосовується вже багато років у UNIX-системах, зображене на рис. 12.6. Під час його опису замість терміна «кластер» вживатимемо його синонім «дисковий блок».

У цьому разі індексний дескриптор містить елементи різного призначення.

- Частина елементів (зазвичай перші 12) безпосередньо вказує на дискові блоки, які називають *прямими* (direct blocks). Отже, якщо файл може вміститися у 12 дискових блоках (за розміру блоку 4 Кбайт максимальний розмір такого файла становитиме 4096 × 12 = 49 152 байти), усі ці блоки будуть прямо адресовані його індексним дескриптором і жодних додаткових структур даних не буде потрібно.
- Якщо файлу необхідно для розміщення даних більше, ніж 12 дискових блоків, використовують непряму адресацію першого рівня. У цьому разі 13-й елемент індексного дескриптора вказує не на блок із даними, а на спеціальний непрямий блок першого рівня (single indirect block). Він містить масив адрес наступних блоків файла (за розміру блоку 4 Кбайт, а адреси – 4 байти в ньому міститимуться адреси 1024 блоків, при цьому максимальний розмір файла буде 4096 × (12 + 1024) = 4 234 456 байт).

- Якщо файлу потрібно для розміщення більше ніж 1024 + 12 = 1036 дискових блоків, використовують непряму адресацію другого рівня. 14-й елемент індексного дескриптора в цьому разі вказуватиме на непрямий блок другого рівня (double indirect block). Такий блок містить масив з 1024 адрес непрямих блоків першого рівня, кожен із них, як зазначалося, містить масив адрес дискових блоків файла. Тому за допомогою такого блоку можна адресувати 1024^2 додаткових блоків.
- Нарешті, якщо файлу потрібно більше ніж $1036 + 1024^2$ дискових блоків, використовують непряму адресацію третього рівня. Останній 15-й елемент індексного дескриптора вказуватиме на непрямий блок третього рівня (triple indirect block), що містить масив з 1024 адрес непрямих блоків другого рівня, даючи змогу адресувати додатково 1024^3 дискових блоків.

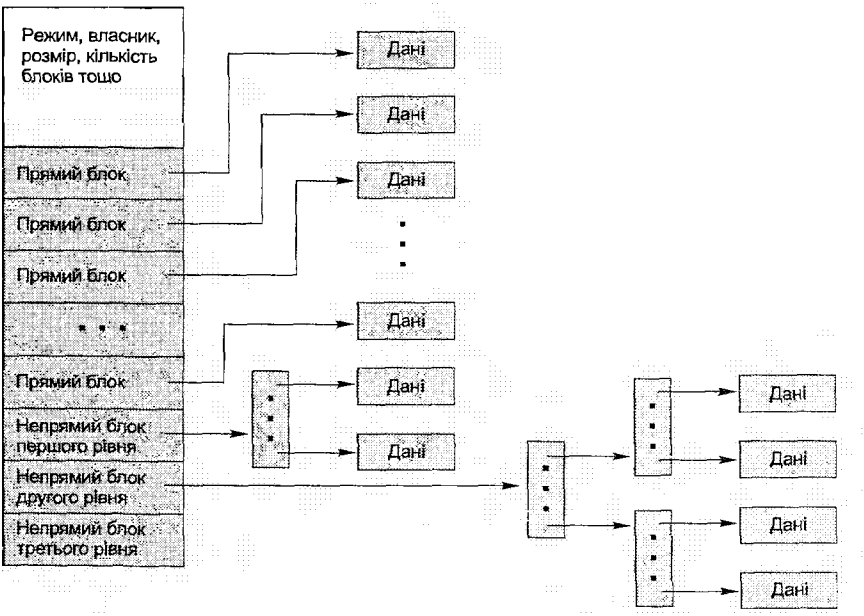

Рис. 12.6. Структура індексного дескриптора в UNIX

Розмір блоку може відрізнятися від 4 Кбайт. Чим більший блок, тим більшим є розмір, що може бути досягнутий файлом, поки не виникне необхідності у непрямій адресації вищого рівня. З іншого боку, більший розмір блоку спричиняє більшу внутрішню фрагментацію.

Можна розподіляти дисковий простір не блоками (кластерами), а їхніми групами (неперервними ділянками із кількох дискових блоків). Такі групи ще називають *екстентами* (extents). Кожен екстент характеризується довжиною (у блоках) і номером початкового дискового блоку. Коли виникає необхідність виділити кілька неперервно розташованих блоків одночасно, замість цього виділяють лише один екстент потрібної довжини. У результаті обсяг службової інформації, яку потрібно зберігати, може бути скорочений.

Характер перетворення адреси в номер кластера робить цей підхід аналогом сторінкової організації пам'яті, причому індексний дескриптор відповідає таблиці сторінок.

Розріджені файли

Багато операційних систем не зберігають покажчики на дискові блоки файлів у їхніх індексних дескрипторах, поки до них не було доступу для записування. Фрагменти, до яких цього доступу не було з моменту створення файла, називають *«дірами»* (holes), дисковий простір під них не виділяють, але під час розрахунку довжини файла їх враховують. У разі читання вмісту «діри» повертають блоки, заповнені нулями, звертання до диска не відбувається.

На практиці «діри» найчастіше виникають, коли покажчик поточної позиції файла переміщують далеко за його кінець, після чого виконують операцію записування. У результаті розмір файла збільшується без додаткового виділення дискового простору. Подібні файли називають *розрідженими файлами* (sparse files). Вони реально займають на диску місця набагато менше, ніж їхня довжина, фактично довжина розрідженого файла може перевищувати розмір розділу, на якому він перебуває.

12.2.6. Організація каталогів

Каталоги звичайно організують як спеціальні файли, що містять набір елементів каталогу (directory entries), кожен з яких відповідає одному файлові.

Елементи каталогу

Елемент каталогу обов'язково містить ім'я файла та інформацію, що дає змогу за іменем файла знайти на диску адреси його кластерів. Структуру такої інформації визначають підходи до розміщення файлів: для неперервного розміщення в елементі каталогу зберігатиметься адреса початкового кластера і довжина файла, для розміщення зв'язними списками – тільки адреса або номер початкового кластера, для індексованого розміщення достатньо зберігати номер індексного дескриптора файла.

Крім обов'язкових даних, елемент каталогу може зберігати додаткову інформацію, характер якої залежить від реалізації. Це може бути, наприклад, набір атрибутів файла (так найчастіше роблять при неперервному розміщенні або розміщенні зв'язаними списками). З іншого боку, за індексованого розміщення всі атрибути файла та іншу службову інформацію зберігають в індексному дескрипторі, а в елемент каталогу додаткову інформацію не заносять (там є тільки ім'я файла і номер дескриптора).

Організація списку елементів каталогу

Реалізація каталогу включає також організацію списку його елементів. Найчастіше елементи об'єднують у лінійний список, але якщо очікують, що в каталогах буде багато елементів, для підвищення ефективності пошуку файла можна використати складніші структури даних, такі як бінарне дерево пошуку або хеш-таблиця. Для прискорення пошуку можна також кешувати елементи каталогу, при цьому під час кожного пошуку файла спочатку перевіряється його наявність у кеші, у разі влучення пошук буде зроблено дуже швидко.

Організація підтримки довгих імен файлів

Розглянемо, яким чином у каталозі зберігають довгі імена файлів. Є ряд підходів до вирішення цієї проблеми.

- Найпростіше зарезервувати простір у кожному елементі каталогу для максимально допустимої кількості символів у імені. Такий підхід можна використати, якщо максимальна кількість символів невелика, у протилежному випадку місце на диску витрачатиметься даремно, оскільки більша частина імен не займатиме весь зарезервований простір.
- Можна зберігати довгі імена в елементах каталогу повністю, у цьому випадку довжина такого елемента не буде фіксованою. Перед кожним елементом каталогу зберігають його довжину, а кінець імені файла позначають спеціальним (зазвичай нульовим) символом. Недоліки цього підходу пов'язані з тим, що через різну довжину елементів виникає зовнішня фрагментація і каталог надто великого розміру може зайняти кілька сторінок у пам'яті, тому під час перегляду такого каталогу є ризик виникнення сторінкових переривань.
- Нарешті, можна зробити всі елементи каталогу однієї довжини, при цьому кожен із них міститиме покажчик на довге ім'я. Усі довгі імена зберігатимуться окремо (наприклад, наприкінці каталогу). Це вирішує проблему зовнішньої фрагментації для елементів каталогу, але питання про керування ділянкою зберігання довгих імен залишається відкритим.

12.2.7. Облік вільних кластерів

Поряд з урахуванням кластерів, виділених для розміщення даних файла, файлові системи мають вести облік вільних кластерів. Це насамперед необхідно для того щоб розв'язати задачу виділення нових кластерів для даних. Для організації керування вільним дисковим простором найчастіше використовують два підходи.

- *Бітовий масив* (бітова карта кластерів), у якій кожен біт відповідає одному кластеру на диску. Якщо відповідний кластер вільний, біт дорівнює одиниці, якщо зайнятий – нулю. Головна перевага такого підходу полягає в тому, що пошук першого ненульового біта можна легко реалізувати, спираючись на апаратну підтримку.
- *Зв'язний список вільних кластерів.* Такий підхід, як зазначалося, найзручніше використати, коли зв'язні списки використовують і для організації розміщення файлів. Звичайно в цьому разі організують список, елементами якого є кластери з адресами (номерами) вільних кластерів на диску.

Достатньо зберігати в пам'яті один елемент списку вільних кластерів або один кластер із бітовою картою. Коли вільні блоки в ньому закінчаться, зчитують наступний елемент. У разі вилучення файла номери його кластерів додають у поточний елемент списку або в поточну бітову карту. Коли місця там більше немає, поточний елемент (карту) записують на диск, а в пам'яті створюють новий елемент або нову карту, куди заносять номери кластерів, для яких забракло місця.

12.3. Продуктивність файлових систем

У цьому розділі розглянемо різні підходи, які використовують для підвищення продуктивності файлових систем. Є дві категорії таких підходів.

- До першої належать різні підходи, які можуть бути впроваджені на етапі проектування і розробки файлової системи: керування розміром блоку, оптимізація розміщення даних, застосування продуктивніших алгоритмів і структур даних. Приклад їхнього застосування наведено у розділі 12.3.1.
- До другої належать низькорівневі універсальні підходи, прозорі для файлової системи (їхнє впровадження може підвищити продуктивність файлової системи без її модифікації). До них належать реалізація дискового кеша і планування переміщення голівок диска. Підходам цієї категорії буде присвячено розділи 12.3.2 і 12.3.3.

12.3.1. Оптимізація продуктивності під час розробки файлових систем

Розглянемо, яким чином можна оптимізувати продуктивність файлової системи зміною структур даних і алгоритмів, які в ній застосовують. У викладі використовуватимемо класичний приклад оптимізації традиційної файлової системи вихідної версії UNIX під час розроблення системи Fast File System (FFS) для BSD UNIX (у наш час ця файлова система також відома як ufs).

Традиційна файлова система UNIX [33, 59] складається із суперблока (що містить номери блоків файлової системи, поточну кількість файлів, покажчик на список вільних блоків), ділянки індексних дескрипторів і блоків даних (рис. 12.7). Розмір блока фіксований і становить 512 байт. Вільні блоки об'єднані у список.

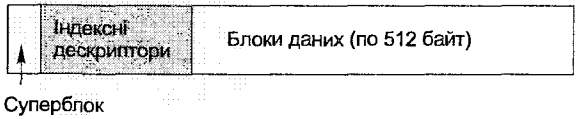

Рис. 12.7. Традиційна файлова система UNIX

Така система є прикладом простого і витонченого вирішення, яке виявилось неприйнятним із погляду продуктивності. На практиці ця файлова система могла досягти на пересиланні даних пропускної здатності, що становить усього 2 % можливостей диска. Назвемо деякі причини такої низької продуктивності.

- Розмір дискового блоку виявився недостатнім, внаслідок чого для розміщення даних файла була потрібна велика кількість блоків; індексні дескриптори навіть для невеликих файлів потребували кількох рівнів непрямої адресації, перехід між якими сповільнював доступ; пересилання даних одним блоком призводила до зниження пропускної здатності.
- Пов'язані об'єкти часто виявлялися віддаленими один від одного і не могли бути зчитані разом, зокрема, індексні дескриптори були розташовані далеко від блоків даних і для каталогу не перебували разом; послідовні блоки для файла також не містилися разом (це траплялося тому, що протягом експлуатації системи через вилучення файлів список вільних блоків ставав «розкиданим»

по диску, внаслідок чого файли під час створення отримували блоки, віддалені один від одного).

До розв'язання цих проблем під час розробки файлової системи FFS були запропоновані декілька підходів [82].

Насамперед, у цій системі було збільшено розмір дискового блока (у FFS звичайно використовували два розміри блока: 4 і 8 Кбайт). Для того щоб уникнути внутрішньої фрагментації (яка завжди зростає зі збільшенням розміру блока), було запропоновано в разі необхідності розбивати невикористані блоки на частини меншого розміру — фрагменти, які можна використати для розміщення невеликих файлів. Мінімальний розмір фрагмента дорівнює розміру сектора диска, звичайно було використано фрагменти на 1 Кбайт.

Крім того, велику увагу було приділено групуванню взаємозалежних даних. З огляду на те, що найбільші втрати часу трапляються під час переміщення головки, було запропоновано розміщувати такі дані в рамках групи циліндрів, яка об'єднує один або кілька суміжних циліндрів. Під час доступу до даних однієї такої групи головку переміщувати було не потрібно, або її переміщення виявлялося мінімальним. Кожна така група за своєю структурою повторювала файлову систему: у ній був суперблок, ділянка індексних дескрипторів і ділянка дискових блоків, виділених для файлів. Тому індексний дескриптор кожного файла розміщувався в тому самому циліндрі, що і його дані, в одній групі циліндрів розміщувалися також всі індексні дескриптори одного каталогу. Послідовні блоки файла прагнули розміщувати в суміжних секторах.

Нарешті, ще одна важлива зміна була зроблена у форматі зберігання інформації про вільні блоки — список вільних блоків було замінено бітовою картою, яка могла бути повністю завантажена у пам'ять. Пошук суміжних блоків у такій карті міг бути реалізований ефективніше. Для ще більшої ефективності цього процесу в системі постійно підтримували деякий вільний простір на диску (коли є вільні дискові блоки, ймовірність знайти суміжні блоки зростає).

Внаслідок реалізації цих і деяких інших рішень пропускна здатність файлової системи зросла в 10–20 разів (до 40 % можливостей диска).

На підставі цього прикладу можна зробити такі висновки:

- розмір блоку впливає на продуктивність файлової системи, при цьому потрібно враховувати можливість внутрішньої фрагментації;
- програмні зусилля, витрачені на скорочення часу пошуку і ротаційної затримки, окупаються (насамперед вони мають спрямовуватися на забезпечення суміжного розміщення взаємозалежної інформації);
- використання бітової карти вільних блоків теж спричиняє підвищення продуктивності.

Ідеї, що лежать в основі FFS, вплинули на особливості проектування файлової системи ext2fs — основної файлової системи Linux, описаної в розділі 13.

12.3.2. Кешування доступу до диска

Найважливішим засобом підвищення продуктивності файлових систем є організація *дискового кеша* (disk cache). Зупинимося докладніше на цьому найважливішому компоненті операційної системи.

Дисковим кешем називають спеціальну ділянку в основній пам'яті, яку використовують для кешування дискових блоків. У разі спроби доступу до дискового блока його поміщають в кеш, розраховуючи на те, що він незабаром буде використаний знову. Під час наступного повторного використання можна обійтися без звертання до диска.

Керування дисковим кешем відбувається на нижчому рівні порівняно з реалізацією файлової системи. Кеш має бути прозорий для файлової системи: блоки, що перебувають у ньому з погляду файлової системи розташовані на диску, відмінності є тільки у швидкості доступу.

Загальний принцип організації дискового кеша показано на рис. 12.8. Для прискорення пошуку потрібного блоку звичайно використовують хеш-функцію, що переводить номер блока у хеш-код. У пам'яті зберігають хеш-таблицю, елементи якої відповідають окремим значенням хеш-коду. Усі блоки з однаковим значенням хеш-коду об'єднують у зв'язні списки. Для пошуку в кеші блока із конкретним номером досить обчислити значення хеш-функції та переглянути відповідний список.

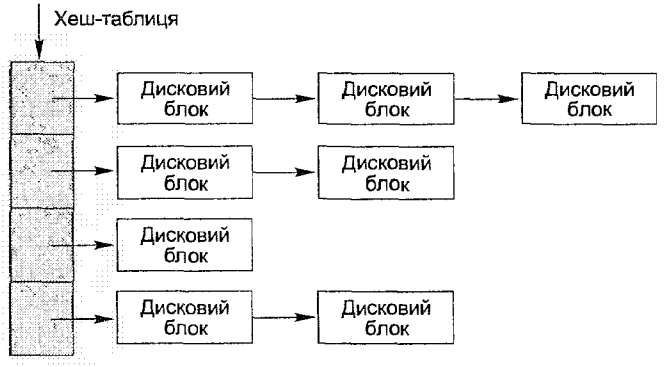

Рис. 12.8. Дисковий кеш

Під час реалізації керування дисковим кешем необхідно отримати відповіді на такі запитання.

- Якого розміру має бути кеш?
- Як має бути організоване заміщення блоків у кеші?
- Як організувати збереження модифікованої інформації із кеша на диск?
- Яким чином оптимізувати завантаження блоків у кеш?

Розмір дискового кеша

Визначаючи розмір дискового кеша, важливо пам'ятати, що він разом із менеджером віртуальної пам'яті з погляду використання основної пам'яті є двома головними підсистемами, що конкурують між собою.

Є два різновиди кеша з погляду керування його розміром: *кеш фіксованого розміру* і *кеш змінного розміру*.

Для кеша фіксованого розміру межі дискового і сторінкового кешів задають жорстко і змінюватися під час виконання вони не можуть. Недоліком такого кеша

є те, що він не може підлаштовуватися під зміну навантаження в системі (наприклад, коли відкрито багато файлів, може знадобитися розширити дисковий кеш, коли завантажено багато процесів – сторінковий).

Стандартна реалізація кеша змінного розміру виділяє спільну пам'ять і під сторінковий, і під дисковий кеш. Під час виконання кожна із підсистем працює з цією пам'яттю, розміщуючи там і сторінки, і дискові блоки (зручно, якщо розмір сторінки дорівнює розміру блока). Недоліком такої реалізації є те, що один файл великого розміру, відкритий процесом, може зайняти блоки, призначені не тільки для відкритих файлів інших процесів, але й для сторінкового кеша віртуальної пам'яті. Для вирішення цієї проблеми можна перейти до змішаного керування кешем, коли межі задають, але їх можна змінити під час виконання.

Заміщення інформації в кеші

Організація заміщення блоків у дисковому кеші багато в чому подібна до реалізації заміщення віртуальної пам'яті. Фактично для визначення заміщуваного блока можна використати більшість алгоритмів заміщення сторінок, про які йшлося в розділі 9, такі як FIFO, LRU, годинниковий алгоритм. Зазначимо, що в цьому разі LRU-алгоритм реалізувати простіше, оскільки звертання до дискового кеша відбувається значно рідше, ніж звертання до пам'яті, і за тривалістю інтервал між такими звертаннями значно перевищує той час, який потрібно затратити на фіксування моменту звертання. Для реалізації LRU-алгоритму можна підтримувати список усіх блоків кеша, упорядкований за часом використання (найстаріший блок – на початку, найновіший – наприкінці), переміщуючи блоки у кінець списку під час звертання до них.

Необхідно враховувати, що за такої ситуації LRU-алгоритм не завжди є кращим або навіть просто прийнятним вирішенням. Наприклад, якщо деякий критично важливий блок не буде записаний на диск після його модифікації, подальший збій може бути фатальним для всієї файлової системи. У той же час у разі використання LRU-алгоритму цей блок буде поміщено в кінець списку, і його записування відкладеться на якийсь час.

Не рекомендують також використовувати алгоритм LRU у численних ситуаціях, коли розмір файла, який зчитують послідовно, перевищує розмір блока. У цьому разі найоптимальнішою є саме зворотна стратегія – MRU (Most Recently Used), коли витісняють блок, що був використаний останнім.

Розглянемо приклад. Нехай файл містить 4 дискові блоки, а в кеші є місце для трьох. Файл зчитують послідовно, при цьому отримують рядок посилань 1234. Якщо використати LRU-алгоритм, то при зчитуванні блоку 4 він заміщує в пам'яті блок 1. Якщо тепер прочитати файл іще раз (отримавши рядок посилань 12341234), побачимо, що першим потрібним блоком буде саме блок 1, який щойно витиснули. Доведеться його зчитувати ще раз, при цьому буде заміщено блок 2, який відразу ж виявиться потрібним знову і т. д. У MRU-алгоритмі блок 4 під час першого читання заміщує блок 3, блоки 1, 2 і 4 під час другого читання перебуватимуть у кеші.

На практиці застосовуються модифіковані схеми LRU, які беруть до уваги категорії важливості блоків, а в разі послідовного доступу до файлів, більших за розміром, ніж блок, зводяться до схеми MRU.

Наскрізне і відкладене записування

З погляду реалізації записування модифікованих даних на диск розрізняють два основні типи дискових кешів: *із наскрізним* (write through cache) і *з відкладеним записом* (write back cache).

Для кеша із наскрізним записом у разі будь-якої модифікації блоку, що перебуває в кеші, його негайно зберігають на диску. Основною перевагою такого підходу (широко розповсюдженого в часи персональних ОС і реалізованого, наприклад, в MS-DOS) є те, що файлова система завжди перебуватиме в несуперечливому стані. Головний недолік цього підходу полягає у значному зниженні продуктивності під час записування даних.

Для кеша із відкладеним записом (такі кеші застосовують у більшості сучасних ОС) під час модифікації блоку його позначають відповідним чином, але на диску не зберігають. Записування модифікованих блоків на диск здійснюється окремо за необхідності (зазвичай у фоновому режимі). Цей підхід значно виграє у продуктивності, але вимагає додаткових зусиль із забезпечення надійності та несуперечності файлової системи. Якщо система зазнає краху в той момент, коли модифіковані блоки ще не були записані на диск, без додаткових заходів (описаних у розділі 12.4) всі зміни втратяться.

Основна проблема, пов'язана із реалізацією відкладеного записування, полягає у виборі ефективного компромісу між продуктивністю і надійністю. Що більший інтервал між збереженням модифікованих даних на диску, то вища продуктивність кеша, але більше роботи потрібно виконати для поновлення даних у разі збою.

Відокремлюють чотири випадки, коли зміни в кеші зберігаються на диску:

- модифікований блок витісняють з кеша відповідно до алгоритму заміщення (аналогічно до того, як це реалізовано для віртуальної пам'яті);
- закривають файл;
- явно видають команду зберегти всі зміни із кеша на диск; в UNIX-системах таку команду звичайно називають sync, вона зводиться до виконання системного виклику із тим самим ім'ям:

```
#include <unistd.h>
sync();    // скидання всіх даних з кеша на диск
```

- проходить заданий проміжок часу (в UNIX-системах за цим стежить спеціальний фоновий процес, який називають update; за замовчуванням такий інтервал для нього становить 30 хвилин).

Системний виклик sync() зберігає всі зміни із кеша на диск. Гнучкішим підходом є збереження змін для конкретних файлів. Для цього у POSIX передбачено системний виклик fsync(). Він блокує виконання процесу до повного збереження всіх буферів відкритого файла із пам'яті на диск.

```
#include <unistd.h>
#include <fcntl.h>
int fd1 = open("myfile.txt", O_RDWR | O_CREAT, 0644);
write(fd1, "hello", sizeof("hello"));
fsync(fd1);            // скидання всіх даних файла з кеша на диск
```

Аналогом fsync() у Win32 API є функція FlushFileBuffers():

```
HANDLE fh = CreateFile("myfile.txt", GENERIC_WRITE, ...);
WriteFile(fh, "hello", sizeof("hello"), ...);
FlushFileBuffers(fh);  // скидання всіх даних файла з кеша на диск
```

Найважливіші блоки (наприклад, блоки з довідковою інформацією, каталогами, атрибутами файлів) варто зберігати якнайчастіше, можливо, навіть після кожної модифікації; блоки із даними можна зберігати рідше.

Для застосувань, які працюють із важливими даними, рекомендовано час від часу робити явне збереження змін на диску (наприклад, виконувати системні виклики sync(), fsync() або їхні аналоги). Для таких застосувань, як СУБД, найкращим вирішенням в основному є повна відмова від дискового кеша (використання *прямого режиму доступу до диска*) і реалізація свого власного кешування. Уже згадувалося про те, що часто такі застосування зовсім відмовляються від послуг файлової системи.

Для реалізації прямого доступу до диска в Linux необхідно під час відкриття файла увімкнути спеціальний прапорець O_DIRECT:

```
int fd1 = open ("myfile.txt", O_RDWR | O_DIRECT, 0644);
```

Аналогом O_DIRECT у Win32 API є FILE_FLAG_WRITE_THROUGH:

```
HANDLE fh = CreateFile("myfile.txt", ...,
FILE_ATTRIBUTE_NORMAL | FILE_FLAG_WRITE_THROUGH, NULL);
```

Випереджувальне читання даних

Під час роботи із диском оптимальною можна вважати ситуацію, коли блоки даних опиняються в пам'яті саме в той час, коли вони мають використовуватись. В ідеальній перспективі (за наявності необмеженої пропускної здатності диска і можливості пророкувати майбутнє) необхідність у кеші може взагалі відпасти – достатньо перед використанням будь-якого блоку просто зчитувати його у пам'ять заздалегідь (або один раз зчитати всі потрібні блоки і потім працювати із ними в пам'яті).

У реальності, як неодноразово було наголошено, знання про майбутнє немає, але можна спробувати його передбачити на підставі аналізу минулого. Таке пророкування може задати стратегію *випереджувального читання даних* (data prefetching, read-ahead).

Під час реалізації випереджувального читання намагаються визначити дискові блоки, які, швидше за все, використовуватимуться разом. Найчастіше при цьому застосовують принцип *просторової локальності* (spatial locality), за яким імовірність спільного використання вища для кластерів, що належать до одного файла і розташовані на диску поруч. При цьому під час випереджувального читання файлова система після читання кластера із номером n перевіряє, чи є в кеші кластер із номером $n+1$, і, якщо він там відсутній, його зчитують у пам'ять заздалегідь, до використання.

Такий підхід спрацьовує для послідовного доступу до файлів, для випадкового доступу він не дає жодного виграшу у продуктивності. За цієї ситуації має сенс «допомагати» системі, організовуючи спільно використовувані дані так, щоб вони на диску перебували поруч. Можна також відслідковувати характер використання файлів і, поки він залишається послідовним, використовувати випереджувальне читання.

Дисковий кеш на основі відображуваної пам'яті

Дотепер ми розглядали так званий традиційний підхід до реалізації дискового кеша, коли система створює окремий пул блоків у пам'яті та працює з ним явно. Такий кеш застосовують у сучасних ОС (насамперед для організації кешування службових даних, що не належать до файлів), але поряд із ним велике значення для реалізації кешування доступу до диска має технологія відображуваної пам'яті.

Зупинимося на особливостях реалізації кеша із використанням цієї технології. У цьому разі в системному адресному просторі резервують спеціальну ділянку, призначену для відображення даних усіх відкритих файлів. При спробі доступу до файла його відображають у цю ділянку (на практиці відображають не весь файл, а деяке його «вікно» фіксованого розміру).

Подальше обслуговування системних викликів доступу до диска здійснюють засобами реалізації відображуваної пам'яті. Під час першого доступу до файла із прикладної програми відбувається спроба скопіювати дані файла із відображеної ділянки у буфер режиму користувача. Оскільки пам'ять у системній ділянці в конкретний момент не виділена, виникають сторінкові переривання, і сторінки починають завантажувати із файла у пам'ять, а потім їхні дані копіюють у буфер користувача. Під час наступних спроб доступу нове відображення не створюється: якщо запит потрапляє у «вікно», використовують уже виділені сторінки (які в цьому разі і є кешем), а якщо запиту немає — нові сторінки завантажують у кеш під час обробки сторінкових переривань.

Цей підхід простий у реалізації, оскільки багато в чому спирається на наявну функціональність менеджера віртуальної пам'яті ОС. Сучасні операційні системи широко його використовують.

12.3.3. Дискове планування

У цьому розділі опишемо ще один підхід до оптимізації доступу до диска, що реалізують на низькому рівні, найчастіше у драйвері диска. Він зводиться до вибору алгоритму, який використовують для планування порядку виконання запитів на переміщення головки диска. Такі алгоритми називають алгоритмами планування переміщення головок або алгоритмами дискового планування.

Алгоритми дискового планування необхідні, бо на один фізичний ресурс (у даному разі маніпулятор диска) припадає багато запитів на використання (у цьому є подібність із алгоритмами планування процесорного часу, де фізичним ресурсом є процесор). Основною метою алгоритмів дискового планування є оптимізація механічних характеристик доступу до диска, насамперед мінімізація часу пошуку (переміщення маніпулятора для позиціювання головки на потрібній доріжці).

Розглянемо три алгоритми дискового планування. Для ілюстрації роботи кожного алгоритму припускатимемо, що спочатку головка перебуває на доріжці 50, після чого їй потрібно виконати запити на звертання до доріжок 90, 190, 40, 120, 0, 130, 70 і 80.

Алгоритм «першим прийшов — першим обслужений»

Найпростішим алгоритмом дискового планування є «першим прийшов — першим обслужений» (First Come – First Served, FCFS, FIFO), коли кожний запит

виконують негайно після його надходження. Цей алгоритм простий у реалізації, справедливий, але недостатньо ефективний. На рис. 12.9 видно, як він спричиняє зайві переміщення головки диска.

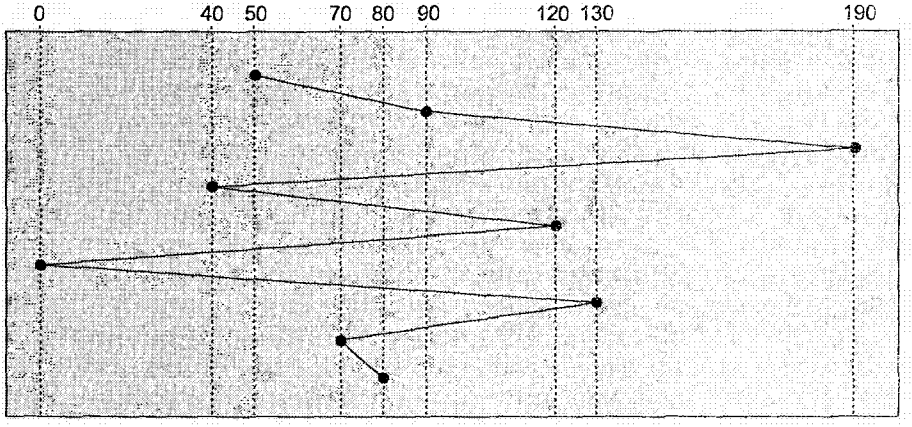

Рис. 12.9. Переміщення головки диска у разі використання алгоритму «першим прийшов — першим обслужений»

Алгоритм «найкоротший пошук — першим»

Цей алгоритм (Shortest Seek Time First, SSTF) планує запити так, щоб першим виконувався той із них, що призводить до мінімального переміщення головки щодо її поточного положення (рис. 12.10). Цей алгоритм набагато ефективніший за FCFS-алгоритм, але не зовсім справедливий – для запитів на переміщення до крайніх доріжок диска він може спричиняти голодування.

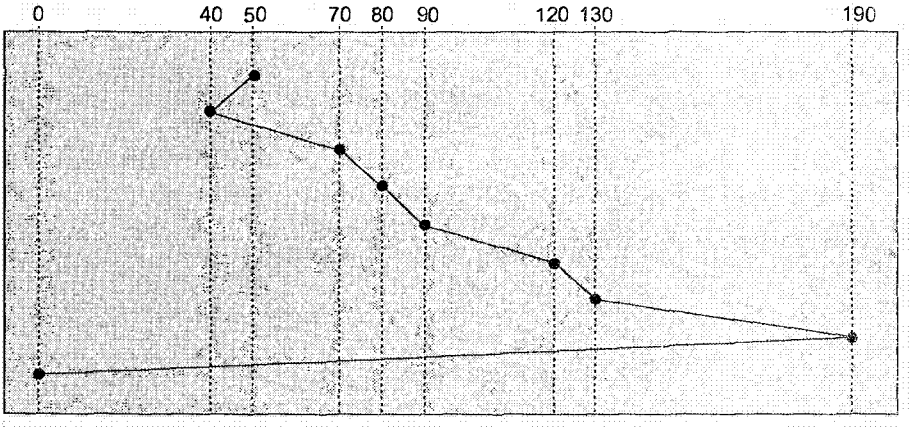

Рис. 12.10. Переміщення головки диска у разі використання алгоритму «найкоротший пошук — першим»

Припустимо, що головка перебуває на доріжці 15 і є запити на переміщення до доріжок 20 і 120. Відповідно до цього алгоритму першим виконують запит на переміщення до доріжки 20. Якщо за цей час у систему надійде новий запит на

переміщення до доріжки 25, наступним буде виконано його, а запит на переміщення до доріжки 120 залишиться чекати далі. Якщо далі постійно надходитимуть запити на переміщення до ближніх доріжок, цей запит може і зовсім не виконатися. У цьому разі отримаємо голодування.

Голодування при використанні цього алгоритму трапляється рідко, але його ймовірність зростає зі збільшенням навантаження.

Алгоритм «ліфта»

Алгоритм «ліфта» (elevator algorithm) використовує той самий принцип, що і ліфт під час переміщення між поверхами. Відомо, коли в кабіні ліфта, що рухається, натиснути кілька кнопок поверхів (як нижче, так і вище від поточного поверху), то вона спочатку відповідатиме на ті з них, що вимагають переміщення в тому напрямку, в якому вона рухалася в момент натискання. Після того як таких запитів не залишиться, кабіна змінить напрямок і почне виконувати запити на переміщення у протилежний бік.

Алгоритм планування переміщає головку диска аналогічно до кабіни ліфта. Спочатку головка рухається у якомусь одному напрямку і виконує ті запити, які вимагають переміщення в цей самий бік. Після того, коли вона доходить до крайньої доріжки, повертає назад і починає виконувати запити на переміщення у зворотному напрямку (рис. 12.11). Далі процес повторюється. Цей алгоритм дещо програє SSTF з погляду середнього часу пошуку, але є справедливішим (голодування тут бути не може).

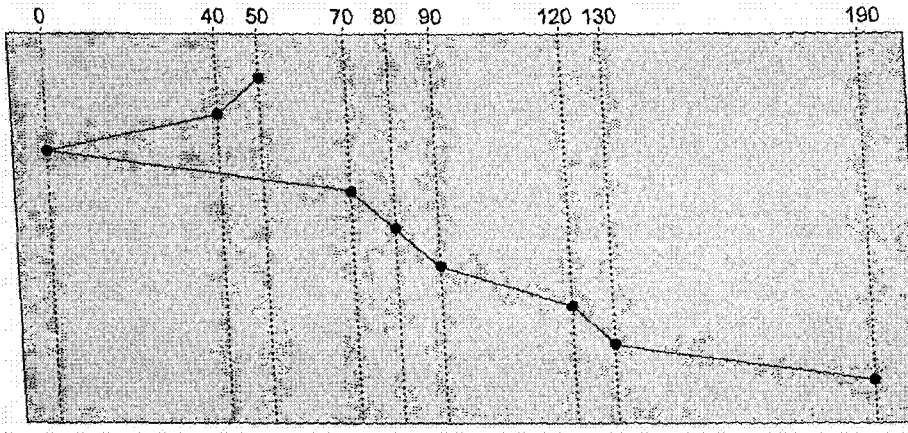

Рис. 12.11. Переміщення головки диска у разі використання алгоритму «ліфта»

12.4. Надійність файлових систем

Цей розділ буде присвячено пошуку відповіді на одне із головних запитань, пов'язаних із функціонуванням файлових систем: що відбудеться із даними файлової системи, коли комп'ютерна система зазнає краху?

Розглянемо, які тут можуть виникнути проблеми.

- У разі використання дискового кеша дані, що перебувають у ньому, після відключення живлення зникнуть. Якщо до цього моменту вони не були записані на диск, зміни в них будуть втрачені. Більш того, коли такі зміни були частково записані на диск, файлова система може опинитися у суперечливому стані.
- Файлова система може опинитися у суперечливому стані внаслідок часткового виконання операцій із файлами. Так, якщо операція вилучення файла видалить відповідний запис із каталогу та індексний дескриптор, але через збій не встигне перемістити відповідні дискові блоки у список вільних блоків, вони виявляться «втраченими» для файлової системи, оскільки розмістити в них нові дані система не зможе.

Щоб забезпечити відновлення після системного збою, можна

- заздалегідь створити резервну копію всіх (або найважливіших) даних файлової системи, аби відновити їх з цієї копії;
- під час першого завантаження після збою дослідити файлову систему і виправити суперечності, викликані цим збоєм (при цьому файлова система має прагнути робити зміни так, щоб обсяг такого дослідження був мінімальним);
- зберегти інформацію про останні виконані операції і під час першого завантаження після збою повторити ці операції.

Як можуть бути реалізовані такі дії, побачимо нижче.

12.4.1. Резервне копіювання

Найвідомішим способом підвищення надійності системи є резервне копіювання даних (data backup). *Резервним копіюванням* або *архівуванням* називають процес створення на зовнішньому носії копії всієї файлової системи або її частини з метою відновлення даних у разі аварії або помилки користувача. Аваріями є вихід жорсткого диска з ладу, фізичне ушкодження комп'ютера, вірусна атака тощо, помилки користувача звичайно зводяться до вилучення важливих файлів. Резервні копії зазвичай створюють на дешевих носіях великого обсягу, найчастіше такими носіями є накопичувачі на магнітній стрічці або компакт-диски.

Одним із головних завдань резервного копіювання є визначення підмножини даних файлової системи, які необхідно архівувати.

- Звичайно створюють резервні копії не всієї системи, а тільки певної підмножини її каталогів. Наприклад, каталоги із тимчасовими файлами архівувати не потрібно. Часто не архівують і системні каталоги ОС, якщо їх можна відновити із дистрибутивного диска.
- Крім того, у разі регулярного створення резервних копій є сенс організувати *інкрементне архівування* (increment backup), коли зберігаються тільки ті дані, які змінились із часу створення останньої копії. Є різні підходи до організації інкрементних архівів. Можна робити повну резервну копію через більший проміжок часу (наприклад, через тиждень), а інкрементні копії – додатково із меншим інтервалом (наприклад, через добу); можна зробити повну копію один раз, а далі обмежуватися тільки інкрементними копіями. Основною проблемою тут є ускладнення процедури відновлення даних.

У момент створення резервної копії важливим є забезпечення несуперечливості файлової системи. В ідеалі резервну копію треба створювати, коли дані файлової системи не змінюються (з нею не працюють інші процеси). На практиці цього домогтися складно, тому використовують спеціальні засоби, які «фіксують» стан системи на деякий момент часу.

Фізичне і логічне архівування

Виділяють два базові підходи до створення резервних копій: *фізичне і логічне архівування*. Під час фізичного архівування створюють повну копію всієї фізичної структури файлової системи, усі дані диска копіюють на резервний носій кластер за кластером. Переваги цього підходу полягають у його простоті, надійності та високій швидкості, до недоліків можна віднести неефективне використання простору (копіюють і всі вільні кластери), неможливість архівувати задану частину файлової системи, створювати інкрементні архіви і відновлювати окремі файли.

Логічне архівування працює на рівні логічного відображення файлової системи (файлів і каталогів). За його допомогою можна створити копію заданого каталогу або інкрементну копію (при цьому відслідковують час модифікації файлів), на основі такого архіву можна відновити каталог або конкретний файл. Зазначимо, що для реалізації коректного відновлення окремих файлів у разі інкрементного архівування необхідно архівувати весь ланцюжок каталогів, що становлять шлях до зміненого файла, навіть якщо жоден із цих каталогів сам не модифікувався із моменту останнього архівування. Стандартну утиліту створення логічних резервних копій в UNIX-системах називають `tar`.

Системне відновлення у Windows XP

Розглянемо приклад організації резервного копіювання на рівні ОС. Служба *системного відновлення* (System Restore) Windows XP дає змогу повертати систему в *точку відновлення* (restore point) — заздалегідь відомий стан, у якому вона перебувала в минулому. За замовчуванням точку відновлення створюють кожні 24 години роботи системи, крім того, можна задавати такі точки явно (наприклад, під час встановлення програмного забезпечення).

Коли служба відновлення створює точку відновлення, формують каталог точки відновлення, куди записують поточні копії системних файлів, після чого спеціальний драйвер системного відновлення починає відслідковувати зміни у файловій системі. Вилучені та змінені файли зберігають у каталозі точки відновлення, інформацію, що описує зміни (назви операцій, імена файлів і каталогів), заносять до журналу відновлення для цієї точки.

Під час виконання операції відновлення (Restore) системні файли копіюють із каталогу точки відновлення у системний каталог Windows, після чого відновлюють змінені файли користувача на підставі інформації із журналу відновлення.

12.4.2. Запобігання суперечливостям і відновлення після збою

Резервне копіювання — це важливий засіб підвищення надійності файлових систем. Однак для файлової системи воно є зовнішнім інструментом. У цьому розділі зупинимося на внутрішніх засобах підвищення надійності. Вони переважно

пов'язані зі зміною алгоритмів виконання файлових операцій і виконанням низькорівневих операцій відновлення.

Методи підвищення надійності файлових систем можуть бути розділені на дві основні групи.

- *Песимістичні* передбачають, що кожна операція роботи із файловою системою може потенційно потерпіти крах, залишивши систему у суперечливому стані. Пропонують модифікувати операції так, щоб цього не допустити. Продуктивність при цьому може знизитися, але надійність зросте. Час відновлення після глобальної аварії буде невеликим.
- *Оптимістичні* передбачають, що операції коли й зазнають невдачі, то рідко, і не варто жертвувати продуктивністю файлової системи для забезпечення надійності кожної операції. Помилки, якщо вони є, залишаються у файловій системі. У разі потреби відновлення після збою запускають процедуру перевірки і відновлення цілісності всієї файлової системи, яка може тривати досить довго. Більшість ОС пропонують утиліти, що здійснюють цю перевірку; в UNIX-системах таку утиліту найчастіше називають `fsck`, у Windows XP – `chkdsk` або `chkntfs`. Необхідно, однак, мати на увазі, що не всі помилки можуть бути виправлені в такий спосіб.

На практиці зазвичай використовують деяку комбінацію цих методів: для деяких помилок (наприклад, тих, які не можуть бути виправлені пізніше) використовують політику запобігання, для всіх інших – оптимістичний підхід із процедурою перевірки.

Несуперечливість у файловій системі

Найпростіше запобігти суперечливостям файлової системи, забезпечивши синхронні операції записування. Ідея проста: виконуючи записування блоку на диск, необхідно дочекатися від диска підтвердження перед тим, як записувати наступний блок.

Насправді такий підхід не може бути застосований для всіх операцій записування (інакше продуктивність знизилася б неприпустимо, наприклад, перестало б працювати кешування). Треба зрозуміти, для якої підмножини операцій його застосування не призводить до неприйнятних результатів. Для цього розглянемо, які суперечливості можуть бути у файловій системі. Як приклад візьмемо індексоване розміщення файлів.

Наведемо кілька правил (інваріантів), які треба виконувати в несуперечливій файловій системі.

- Усі вільні блоки мають перебувати в списку вільних блоків (і навпаки, всі елементи списку вільних блоків мають справді бути вільними).
- Дисковий блок має бути використаний тільки одним файлом (два індексних дескриптори не можуть вказувати на один і той самий блок).
- Лічильник жорстких зв'язків файлового дескриптора має збігатися із числом жорстких зв'язків, що справді посилаються на нього.

Розглянемо проблеми, що виникають у разі порушення цих інваріантів.

Некоректний список вільних блоків

Для ілюстрації цієї проблеми почнемо з послідовності кроків під час виконання операції створення файла.

1. Шукають у поточному робочому каталозі файл із тим самим ім'ям. Якщо він є, повертають помилку, у протилежному випадку відшукують місце для елемента каталогу.
2. Шукають вільний індексний дескриптор. Знайдений дескриптор позначають як виділений.
3. Ім'я та номер дескриптора додають в елемент каталогу.

У разі збою між кроками 2 і 3 дескриптор залишиться позначений як виділений, але фактично таким не буде. У результаті з'являються втрачені дані.

Проблема втрачених даних виникає під час ситуації, коли невикористані ресурси позначають як виділені, звичайно це відбувається за наявності списку або карти вільних блоків. Для запобігання її появі потрібно дотримуватися такого правила: не можна постійно зберігати покажчик на об'єкт, що перебуває у списку вільних блоків (як це зробили на кроці 2).

Подібна проблема виникає і під час вивільнення блоків. Як приклад можна навести операцію скорочення файла, коли довжину файла скорочують, а зайві блоки вивільняють. Її можна виконувати так.

1. Покласти покажчик на блок в індексному дескрипторі рівним нулю.
2. Помістити блок у список вільних блоків.

Якщо поміняти місцями кроки 1 і 2 (спочатку помістити блок у список вільних блоків, а потім вилучити із дескриптора), то збій між цими діями спричиняє невірне припущення, що блок вивільнений, хоча фактично вільним він не є (на нього вказує індексний дескриптор).

Тому можна сформулювати друге правило, протилежне до першого: повторно використати ресурс можна тільки після того, як були обнулені всі покажчики на нього.

Для відновлення списку вільних блоків після збою можна застосовувати й оптимістичний підхід, реалізований утилітою типу `fsck`. Для цього використовують збирання сміття із маркуванням і очищенням (див. розділ 10.6.2). Спочатку припускають, що список вільних блоків включає всі блоки диска (якщо він реалізований як бітова карта, це зробити легко); це буде етап маркування. Після цього потрібно почати із кореневого каталогу і рекурсивно обійти всі підкаталоги, вилучаючи всі їхні блоки зі списку вільних блоків — це буде етап очищення. Цей алгоритм одночасно відновлює виділені блоки, позначені як вільні, та втрачені блоки. Його недоліком є те, що потрібен обхід усього дерева каталогів (для великих файлових систем це може займати багато часу).

Некоректні значення лічильника зв'язків

Для ілюстрації цієї проблеми розглянемо послідовність кроків під час виконання операції вилучення жорсткого зв'язку для файла (unlink).

1. Здійснюють обхід каталогу в пошуку цього імені. Якщо його не знайдено, повертають помилку.
2. Очищають елемент каталогу.

3. Зменшують лічильник жорстких зв'язків індексного дескриптора.
4. Якщо лічильник жорстких зв'язків дорівнює нулю, очищають індексний дескриптор і всі блоки, на які він вказує.

У разі збою між кроками 2 і 3 буде отримано надто велике значення лічильника зв'язків (зв'язок вилучений, а лічильник не змінився). Блоки, що належать до цього файла, ніколи не вивільняться (оскільки лічильник зв'язків тепер ніколи не досягне нуля). Для великих файлів це може спричинити істотні втрати дискового простору.

Припустимо, що очищення елемента каталогу і зменшення лічильника поміняли місцями (тобто спочатку зменшують лічильник, а потім вилучають зв'язок). Тепер, якщо збій відбудеться між цими двома операціями, буде надто мале значення лічильника зв'язків (коли зв'язків стане більше, ніж значення лічильника). Ця проблема ще серйозніша, ніж попередня, оскільки використовувані блоки будуть позначені як вільні. Є багато різних проявів цієї проблеми: від «зниклого» вмісту файлів (що перейшов до іншого файла) до переміщення у вільні блоки вмісту системних файлів, наприклад, файла паролів.

Є два способи боротьби з цією проблемою.

1. Оптимістичний – обходити дерево каталогів і коригувати значення лічильника зв'язків для кожного індексного дескриптора в рамках утиліти типу `fsck`.
2. Песимістичний – ніколи не зменшувати лічильник зв'язків до обнулення покажчика на об'єкт (наприклад, завжди синхронно зберігати на диску список вільних блоків під час кожного розміщення та вивільнення блока). При цьому використовують 1-2 додаткові операції записування на диск.

Приклад організації боротьби із суперечливостями у FFS

На завершення наведемо приклад реалізації описаних раніше підходів у вже знайомій файловій системі FFS.

Насамперед зазначимо, що ця файлова система поєднує оптимістичний і песимістичний підходи до реалізації суперечливості.

З одного боку, вона використовує утиліту `fsck` для корекції списку вільних блоків і значень лічильника зв'язків відповідно до описаних раніше алгоритмів.

З іншого боку, реалізація песимістичних підходів до суперечності ґрунтується на виконанні двох інваріантів.

1. Усі імена в каталогах завжди посилаються на коректні індексні дескриптори.
2. Жоден блок не використовується більш як одним індексним дескриптором.

Підтримку справедливості цих інваріантів реалізують на основі виконання трьох правил.

1. Новий індексний дескриптор зберігають на диску до того, як ім'я зв'язку додають у каталог.
2. Ім'я зв'язку вилучають із каталогу до вивільнення індексного дескриптора.
3. Вивільнений індексний дескриптор зберігають на диску до того, як його блоки помістять у список вільних блоків.

У результаті створення і вилучення файла необхідні по дві синхронні операції обміну із диском.

Ще однією властивістю FFS є можливість пошуку втрачених індексних дескрипторів. Вони можуть бути втрачені, коли каталог буде ушкоджено або відбудеться збій до встановлення жорсткого зв'язку. Такий пошук ґрунтується на двох особливостях реалізації системи.

- Індексні дескриптори заздалегідь розміщують у відомих місцях на диску. У результаті місцезнаходження кожного дескриптора може бути знайдене незалежно від того, є на нього покажчики чи ні.
- Вміст вільних індексних дескрипторів заповнюють нулями. Із цього система може зробити висновок, що будь-який дескриптор з ненульовим вмістом дотепер використовують. Під час обходу `fsck` поміщає дескриптори з ненульовим вмістом, для яких не задано жодного зв'язку, у спеціальний каталог **/lost+found**.

Зазначимо, що більшість названих вирішень використовують і у файловій системі ext2fs для Linux.

12.4.3. Журнальні файлові системи

Великий обсяг дисків робить виконання програми перевірки і відновлення під час завантаження після збою досить тривалим процесом (для диска розміром у десятки Гбайтів така перевірка може тривати кілька годин). У деяких ситуаціях (наприклад, на серверах баз даних з оперативною інформацією) подібні затримки із відновленням роботоздатності системи після кожного збою можуть бути недопустимими. Необхідно організувати збереження інформації таким чином, щоб відновлення після збою не вимагало перевірки всіх структур даних на диску. Спроби розв'язати цю проблему привели до виникнення *журнальних файлових систем* (logging file systems).

Основна мета журнальної файлової системи — надати можливість після збою, замість глобальної перевірки всього розділу, робити відновлення на підставі інформації *журналу* (log) — спеціальної ділянки на диску, що зберігає опис останніх змін. Використання журналу засноване на важливому спостереженні: під час відновлення після збою потрібно виправляти тільки інформацію, яка перебувала у процесі зміни у момент цього збою. Це та інформація, що не встигла повністю зберегтися на диску (зазвичай вона займає тільки малу його частину).

Основна ідея таких файлових систем — виконання будь-якої операції зміни даних на диску у два етапи.

1. Спочатку інформацію зберігають у журналі (у ньому створюють новий запис). Таку операцію називають *випереджувальним записуванням* (write-ahead) або *веденням журналу* (journaling).
2. Коли ця операція повністю завершена (було підтверджено зміну журналу), інформацію записують у файлову систему (можливо, не відразу). Після того, як зміну журналу було підтверджено, усі записи в журналі, створені на етапі 1, стають непотрібними і можуть бути вилучені. Зауважимо, що синхронізація журналу і реальних даних на диску може відбуватися і явно; виконання такої операції називають *точкою перевірки* (checkpoint). Дані із журналу після цієї перевірки теж можуть бути вилучені.

Читання даних завжди здійснюють із файлової системи, журнал у цій операції не бере участі ніколи.

Після того, як інформація про зміни потрапила в журнал, самі ці зміни можуть бути записані у файлову систему не відразу. Часто об'єднують кілька операцій зміни даних у файловій системі, якщо вони належать до одного кластера, для того щоб виконати їх усі разом. У результаті кількість операцій звертання до диска істотно знижується. Ще однією важливою обставиною підвищення продуктивності, є той факт, що операції записування в журнал виконують послідовно і без пропусків. Найкращої продуктивності можна домогтися, помістивши журнал на окремий диск.

Розмір журналу має бути достатній для того, щоб у ньому помістилися ті зміни, які на момент збою можуть перебувати у пам'яті.

Є різні підходи до того, яка інформація має зберігатися в журналі.

- Тільки описи змін у метаданих (до метаданих належить вся службова інформація: індексні дескриптори, каталоги, імена тощо). Такий журнал забезпечує несуперечливість файлової системи після відновлення, але не гарантує відновлення даних у файлах. З погляду продуктивності це найшвидший спосіб.
- Змінені кластери повністю. Такий підхід не вирізняється високою продуктивністю, натомість з'являється можливість відновити дані повністю.

Файлові системи звичайно дають змогу вибрати варіант збереження інформації в журналі (це може бути зроблено під час монтування системи). На практиці вибір підходу залежить від конкретної ситуації.

Програма відновлення файлів має розрізняти дві ситуації.

- Збій відбувся до підтвердження зміни журналу. У цьому разі здійснюють *відкат* (rollback): цю зміну ігнорують, і файлова система залишається в несуперечливому стані, у якому вона була до операції. Зазначимо, що такі атомарні операції мають багато спільного із *транзакціями* — атомарними операціями у базі даних (відомо, що сервери баз даних для підтримки транзакцій також реалізують роботу із журналом).
- Збій відбувся після підтвердження зміни журналу. За цієї ситуації потрібно відновити дані на підставі інформації журналу (такий процес ще називають *відкатом уперед* — rolling forward).

Журнал транзакцій у базі даних дає змогу підтверджувати і скасовувати визначений користувачем набір операцій зміни даних, що і є транзакцією. На відміну від цього, підтвердження і відкат на підставі інформації журналу файлової системи можуть зачіпати тільки результати виконання окремих системних викликів. Припустимо, що відбувається копіювання великого файла частинами, при цьому алгоритм копіювання в циклі звертається до системного виклику `write()` для копіювання кожної частини. Коли під час такого копіювання відбудеться збій між окремими викликами `write()`, то за будь-яких стратегій реалізації журналу файл залишиться скопійованим неповністю — не можна підтверджувати або скасовувати кілька викликів як єдине ціле.

Сучасні операційні системи все більше переходять до використання журнальних файлових систем; наприклад, для Linux є декілька їх реалізацій (ext3fs, ReiserFS, XFS). Файлова система NTFS також підтримує ведення журналу.

Висновки

- Фізичне розміщення даних у файловій системі має забезпечувати ефективність доступу. Для цього необхідно враховувати механіку сучасних дискових пристроїв, приділяючи основну увагу мінімізації часу пошуку і ротаційної затримки.
- Основними підходами до фізичного розміщення даних є неперервне розміщення, розміщення зв'язними списками (важливою модифікацією якого є використання FAT) та індексоване розміщення. Каталоги звичайно реалізовані як спеціальні файли.
- Для підвищення продуктивності файлової системи в деяких випадках необхідно змінити її структуру (застосувати ефективні алгоритми розміщення, структури даних тощо). Такі зміни вимагають доступу до вихідного коду цієї системи і можуть бути вироблені тільки її розробниками.
- До підвищення продуктивності може також привести реалізація таких вирішень, як дисковий кеш або дискове планування. Ці вирішення є прозорими для файлової системи. Вони звичайно реалізовуються на нижчому рівні, наприклад, на рівні дискових драйверів.
- Для підвищення надійності файлових систем можна використати високорівневі підходи, такі як організація резервного копіювання, і підходи низького рівня (запобігання суперечностям внаслідок додаткових дій під час кожної файлової операції, відновлення системи після збоїв).
- Журнальні файлові системи дають змогу відновлювати дані на підставі інформації журналу, в якому зберігають відомості про операції, виконані з файловою системою. Цю інформацію зберігають перед остаточним записом даних на диск.

Контрольні запитання та завдання

1. Порівняйте продуктивність файлових систем на основі FAT і з розміщенням зв'язними списками для послідовного і випадкового доступу до файла.
2. Припустімо, що замість використання індексних дескрипторів розробник файлової системи реалізував збереження усієї відповідної інформації в елементі каталогу. Які проблеми при цьому виникають?
3. Припустимо, що у файловій системі підтримується збереження кількох версій файла. Де і як потрібно зберігати інформацію про ці версії? Як повинні виглядати базові системні виклики для роботи з файлами в такій системі?
4. Індексний дескриптор містить інформацію про атрибути файла (16 байт), 13 прямих блоків і по одному непрямому блоку трьох рівнів. Вільне місце у дескрипторі також може бути зайняте даними файла. Обчисліть максимальний розмір файла, якщо розмір дискового блоку дорівнює 512 байт, а розмір адреси блоку — 4 байти. Скільки операцій звертання до диска необхідно виконати, щоб прочитати дисковий блок у позиції 300 000 з використанням такого індексного дескриптора?

5. На скільки збільшиться максимальний розмір файла для індексованого розміщення у разі:
   а) збільшення розміру блоку вдвічі;
   б) введення четвертого рівня дотичності?
6. Чи є обов'язковою наявність у системі списку вільних блоків? Як створити заново такий список у випадку його втрати?
7. Під час виконання тестового набору процесів у системі з дисковим кешем фіксованого розміру було виявлено, що звертань до диска досить багато. Після збільшення обсягу кеша продуктивність системи під час виконання цього набору процесів ще більше знизилася. Поясніть, чому це відбулося.
8. Назвіть проблеми, що можуть виникнути у випадку, якщо менеджер кеша і менеджер віртуальної пам'яті спільно використовують одну й ту саму фізичну пам'ять, фактично будучи конкурентами. У яких ситуаціях додавання основної пам'яті для використання менеджером віртуальної пам'яті дасть більший виграш у продуктивності порівняно зі збільшенням кеша? У яких ситуаціях, навпаки, вигідніше збільшувати кеш?
9. Запити до диска на переміщення голівки пристрою приходять у такому порядку: 10, 22, 20, 2, 40, 6, 38. Швидкість переміщення голівки — 6 мс/доріжку. У поточний момент голівка перебуває на доріжці 20. Нумерація доріжок — від 0 до 49. Обчисліть час пошуку у разі застосування алгоритмів:
   а) FIFO;
   б) SSTF;
   в) алгоритму ліфта (голівка починає рух у напрямку від доріжки 0).
10. Перелічіть можливі варіанти поведінки системи резервного копіювання у випадку виявлення нею символічного зв'язку. Який варіант найкращий?
11. Поясніть, чому в системі FFS індексний дескриптор записують на диск, перед тим як зберегти на диску іншу інформацію з новою файла.
12. Утиліта `fsck` виконує повний обхід дерева каталогів системи з індексованим розміщенням файлів і формує списки вільних і використовуваних дискових блоків. Яка проблема наявна у файловій системі, якщо:
    а) ім'я одного й того самого блоку є в обох списках;
    б) імені існуючого блоку немає в жодному з них?
    Які дії може виконати утиліта в тому й іншому випадку?
13. У UNIX-системах багаторазово розглядали і щоразу відкидали пропозицію дозволити встановлення жорстких зв'язків для каталогів. Чому?
14. Опишіть фактори, що впливають на вибір розміру журналу в журнальній файловій системі. За яких умов журнальна файлова система може виявитися продуктивнішою порівняно з такою самою системою, але без журналу?
15. Для яких операцій з журнальною файловою системою:
    а) у журнал записують більше даних, ніж безпосередньо у файли;
    б) з журналу зчитують більше даних, ніж із файлів?

Розділ 13

Реалізація файлових систем

- Інтерфейс файлової системи VFS
- Файлові системи Linux: ext2fs, ext3fs і /proc
- Файлові системи лінії FAT
- Файлова система NTFS
- Системний реєстр Windows XP

У цьому розділі нами буде розглянуто деякі приклади реалізації конкретних файлових систем.

13.1. Інтерфейс віртуальної файлової системи VFS

Більшість UNIX-систем сьогодні керують різними типами файлових систем із використанням універсального рівня програмного забезпечення, який називають *віртуальною файловою системою* (Virtual File System, VFS). Така система надає прикладним програмам однаковий програмний інтерфейс, що реалізується за допомогою системних викликів роботи з файлами, описаних у розділі 11.5; надає розробникам файлових систем набір функцій, які їм треба реалізувати для інтеграції їхньої системи в інфраструктуру VFS (цей набір функцій називають *інтерфейсом файлової системи*).

Загальні принципи організації VFS на прикладі її реалізації в Linux описані нижче.

13.1.1. Основні функції VFS

Основною метою VFS є забезпечення можливості роботи ОС із максимально широким набором файлових систем. Цей рівень програмного забезпечення перетворює стандартні системні виклики UNIX для керування файлами у виклики функцій низького рівня, реалізованих розробником конкретної файлової системи.

Рівень VFS забезпечує доступ через стандартні файлові системні виклики до будь-якого рівня програмного забезпечення, що реалізує інтерфейс файлової системи. Це програмне забезпечення може взагалі не працювати із дисковою файловою системою, а, наприклад, генерувати всю інформацію «на льоту». Програмні модулі, що реалізують інтерфейс файлової системи, називаються модулями підтримки файлових систем.

Файлові системи, підтримувані VFS, можуть бути розділені на три основні категорії.

- *Дискові* є файловими системами в їхньому традиційному розумінні (розташовані на диску). Вони можуть мати будь-яку внутрішню структуру, важливо тільки, щоб відповідне програмне забезпечення реалізовувало інтерфейс файлової системи. Серед дискових файлових систем, які підтримує VFS, можна виділити розроблені спеціально для Linux (ext2fs, ext3fs, ReiserFS); Windows XP (лінія FAT, NTFS); інших версій UNIX (FFS, XFS); CD-ROM (ISO9660).
- *Мережні* реалізують прозорий доступ до файлів на інших комп'ютерах через мережу. До них належать NFS (забезпечує доступ до інших UNIX-серверів) і SMB (виконує аналогічні функції для серверів мережі Microsoft).
- *Спеціальні* або *віртуальні* відображають у вигляді файлової системи те, що насправді файловою системою не є. Вони не керують дисковим простором ні локально, ні віддалено. Прикладом віртуальної файлової системи є файлова система /proc, про яку йтиметься у розділі 13.3.

Розглянемо, як працює VFS на прикладі копіювання файла із дискети на жорсткий диск за допомогою коду, наведеного у розділі 11.5.2. На дискеті розміщена файлова система FAT, на диску – стандартна для Linux файлова система ext2fs. Обидва ці дискових пристрої монтуються в окремі каталоги дерева каталогів UNIX. Припустимо, що шлях до файла на дискеті буде /floppy/src.txt, шлях до файла на диску – /tmp/dest.txt.

Інфраструктура VFS абстрагує відмінності між файловими системами джерела і місця призначення (рис. 13.1).

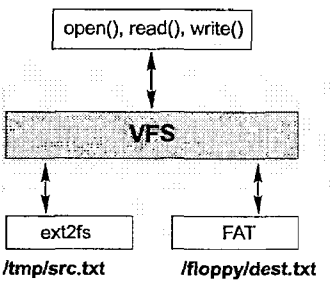

Рис. 13.1. Участь VFS в операції копіювання файла

Прикладна програма, що здійснює копіювання, має зробити тільки такі універсальні дії:

```
// відкрити файл на дискеті
infile = open ("/floppy/src.txt", O_RDONLY);
// створити новий файл на жорсткому диску
outfile = open ("/tmp/dest.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);
// подальший код аналогічний до прикладу в розділі 11.5.2
```

Як видно, жоден із викликів не використовує інформацію, специфічну для тієї чи іншої файлової системи. Для реалізації зворотного копіювання (із диска на дискету) досить поміняти місцями імена файлів – увесь інший код залишиться незмінним.

13.1.2. Загальна організація VFS

Під час розробки VFS були широко використані принципи об'єктної орієнтації. Ця інфраструктура складається із двох основних груп елементів: набору правил, яким мають підлягати файлові об'єкти, і рівня програмного забезпечення для керування цими об'єктами. Базова архітектура VFS визначає три основних об'єктних типи.

- *Об'єкт індексного дескриптора* (об'єкт `inode`, inode object) описує набір атрибутів і методів, за допомогою яких відображують файл на рівні файлової системи. Його назва свідчить про те, що більшість файлових систем в UNIX використовує індексоване розміщення даних; насправді цей об'єкт може інкапсулювати будь-яку структуру фізичного розміщення файла на диску.
- *Об'єкт відкритого файла* (об'єкт `file`, file object) відображає відкритий файл на рівні процесу.
- *об'єкт файлової системи* (filesystem object) відображає всю файлову систему; у Linux його називають *об'єктом суперблока* (superblock object).

У реалізації VFS у Linux до цих трьох об'єктів додають ще один: об'єкт елемента каталогу (об'єкт `dentry`, dentry object), що відображає елемент каталогу і його зв'язок із файлом на диску.

Для кожного із цих типів об'єктів VFS визначає набір операцій (методів), які мають бути реалізовані в об'єктах. Набір цих методів становить інтерфейс файлової системи. У кожному об'єкті є покажчик на таблицю набору адрес функцій, що реалізують ці методи для конкретного об'єкта. Програмне забезпечення VFS завжди викликає ці функції через покажчики на них, йому не потрібно знати заздалегідь, із яким конкретним типом об'єкта воно працює (з погляду об'єктного підходу у VFS реалізовано поліморфізм). Так, під час читання даних через об'єкт файлового дескриптора для рівня VFS байдуже, який саме елемент конкретної файлової системи відображає цей об'єкт (такими елементами можуть бути мережні файли, дискові файли різних файлових систем тощо). На цьому рівні відбувається тільки виклик стандартного методу читання даних через покажчик на нього. Реальна файлова система може зовсім не використовувати індексних дескрипторів — відповідний об'єкт генеруватиметься «на ходу».

13.1.3. Об'єкти файлової системи

Поняття об'єкта файлової системи

Об'єкт файлової системи (об'єкт суперблоку в Linux) відображає пов'язаний набір файлів, що міститься в ієрархії каталогів. Ядро підтримує по одному такому об'єкту для кожної змонтованої файлової системи. У Linux об'єкт суперблока відображений структурою `super_block`.

Основне завдання такого об'єкта — надання доступу до об'єктів індексних дескрипторів. Кожний із них у VFS ідентифікують унікальною парою (файлова система, номер індексного дескриптора); якщо виникає потреба отримати доступ до індексного дескриптора за його номером, програмне забезпечення VFS звертається до об'єкта файлової системи, який і повертає відповідний до цього номера об'єкт індексного дескриптора.

Монтування файлової системи

Об'єкт файлової системи, крім того, надає операції над файловою системою загалом. Насамперед це стосується системного виклику mount(), використовуваного для монтування файлової системи в каталог іншої файлової системи:

```
#include <sys/mount.h>
int mount(const char *partition, const char *dir, const char *fstype,
unsigned long mflags, const void *data );
```

де: partition — розділ, який необхідно монтувати (задають у вигляді імені спеціального *файла пристрою* в каталозі /dev; про них ітиметься в розділі 15);

dir — каталог, у який монтуватиметься розділ;

fstype — тип файлової системи (задають у вигляді рядка символів, наприклад: "ext2" для ext2fs, "msdos" для FAT, "iso9660" для файлової системи CD-ROM);

mflags — набір прапорців режиму монтування, наприклад MS_RDONLY для монтування системи тільки для читання).

Наведемо приклад виклику для монтування першого розділу жорсткого диска (заданого файлом /dev/hda1) із файловою системою ext2fs у каталог /usr:

```
mount("/dev/hda1", "/usr", "ext2", 0, NULL);
```

Для розмонтування файлової системи використовують системний виклик umount():

```
umount("/usr");
```

Отримання інформації про файлову систему

Для отримання інформації про змонтовану файлову систему через інтерфейс VFS у Linux використовують системний виклик statfs():

```
#include <sys/vfs.h>
int statfs(const char *path, struct statfs *fsinfo);
```

де: path — шлях до будь-якого файла на цій файловій системі (якщо файл відсутній, виклик поверне -1, і структура не заповниться);

fsinfo — покажчик на структуру типу statfs, куди будуть занесені дані про файлову систему.

Серед полів структури statfs можна виокремити такі (усі типу long):

- f_blocks — загальна кількість блоків на файловій системі;
- f_bavail — кількість вільних блоків, доступних звичайному користувачу;
- f_files — загальна кількість індексних дескрипторів;
- f_ffree — кількість вільних індексних дескрипторів.

Наведемо приклад виклику:

```
struct statfs fs;
if (statfs("./myfile.txt", &fs) == -1) { printf("помилка\n"); exit(-1); }
printf("Усього блоків %d, доступно %d\n", fs.f_blocks, fs.f_bavail);
```

13.1.4. Доступ до файлів із процесів

Зупинимося на засобах VFS, що забезпечують організацію доступу до файлів із процесів. Такі засоби є найважливішим елементом цієї інфраструктури.

Об'єкти файлових дескрипторів і відкритих файлів

Спочатку розглянемо об'єкти `inode` і `file`. Обидва ці об'єкти використовують для доступу до окремого файла. Об'єкт файлового дескриптора відображає файл як ціле, тоді як об'єкт відкритого файла зображає точку доступу до даних усередині файла. Процес зазвичай не може одержати доступ до вмісту об'єкта `inode` без попереднього доступу до пов'язаного із ним об'єкта `file`.

Об'єкт `inode` відображає файловий дескриптор відповідної файлової системи. Він містить різну інформацію, специфічну для логічного та фізичного відображення файла файлової системи, зокрема номер відповідного індексного дескриптора на диску, кількість блоків, інформацію про модифіковані блоки, атрибути файла, його розмір.

Об'єкт `file` створюють за допомогою системного виклику відкриття файла (`open`). В об'єкті зберігається режим доступу до файла (читання або записування, поле `f_mode`) і покажчик його поточної позиції (поле `f_pos`). Такий об'єкт, крім того, забезпечує організацію випереджувального читання даних на підставі дій, виконаних процесом. Йому не відповідають дані реальної файлової системи. Усі використовувані об'єкти `file` файлової системи містяться у двозв'язному списку об'єктів відкритих файлів (`s_files`).

Об'єкти `file` в основному належать процесу, що їх відкрив, водночас об'єкти `inode` від процесів не залежать — до одного такого об'єкта можуть звертатися кілька процесів. Справді, навіть якщо до файла не звертається жоден процес, відповідний об'єкт `inode` може зберігатися системою (у кеші індексних дескрипторів — inode cache) для підвищення продуктивності, у разі коли файл буде використаний у найближчому майбутньому. Є низка структур даних, які використовуються для організації об'єктів `inode` у системі, серед них список невикористовуваних індексних дескрипторів (що працюють як кеш), два списки використовуваних дескрипторів — модифікованих і чистих, а також хеш-таблиця, що може прискорити пошук у ситуації, коли відомі номер індексного дескриптора і файлова система.

Для реалізації об'єкта `inode` розробник файлової системи має реалізувати набір методів, серед яких є `create()`, `link()`, `unlink()` і т. д.

Відзначимо відмінності в обробці каталогів. Деякі операції, визначені над каталогами (наприклад, `lookup()`, `mkdir()`, `rmdir()` тощо), не потребують попереднього відкриття відповідного каталогу як файлу (створення об'єкта `file`); ці методи реалізовані безпосередньо в об'єкті `inode`.

Об'єкти елементів каталогу

Крім інформації про файли на диску та їхнє використання процесами система має зберігати відомості про елементи каталогу, відповідальні за перетворення імен файлів у індексні дескриптори. Це, насамперед, потрібно для того, щоб організувати кешування часто використовуваних елементів каталогу. Під час кешування для них зберігають інформацію про те, який елемент відповідає певному індексному дескриптору. За наявності елемента каталогу в кеші немає потреби переглядати диск у пошуках дескриптора, що може дати досить значний виграш у швидкості.

У Linux ці міркування привели до введення ще однієї категорії об'єктів VFS — елементів каталогу (dentry objects). Ці об'єкти розташовані між об'єктами відкритих файлів і об'єктами індексних дескрипторів. Кожний такий об'єкт містить інформацію про елемент каталогу (насамперед його ім'я і посилання на об'єкт

відповідного індексного дескриптора). Кілька об'єктів dentry можуть посилатися на один об'єкт inode, якщо вони відповідають жорстким зв'язкам (різним елементам каталогів, що посилаються на один індексний дескриптор).

Об'єкти dentry, як і об'єкти inode, існують незалежно від процесів. Коли кілька процесів відкривають один і той самий файл через один і той самий елемент каталогу, для них не створюють окремі об'єкти dentry; замість цього всі їхні об'єкти file посилаються на один такий об'єкт, що відображає цей елемент.

Після використання об'єкти dentry поміщають у кеш об'єктів елементів каталогу (dentry cache). Він є одним із кешів кускового розподілювача пам'яті. Під час доступу до елемента каталогу спочатку завжди перевіряють, чи немає відповідного об'єкта dentry у цьому кеші; якщо він там є, можна негайно отримати доступ до відповідного індексного дескриптора.

Під час пошуку місцезнаходження файла, заданого повним шляхом, цей шлях розбивають на частини, що відповідають каталогам та імені файла, після чого кожна частина відображається окремим об'єктом dentry. Кожну із частин у результаті можна отримати з кеша, що дає змогу прискорити пошук.

Зв'язок процесу із об'єктами VFS

Тепер розглянемо, яким чином процес отримує доступ до об'єктів відкритих файлів. У керуючому блоці кожного процесу є два поля, що задають зв'язок цього процесу із файловою системою.

Перше з них – поле fs, що задає покажчик на структуру типу fs_struct. Вона містить поля, що відповідають елементам файлової системи, які задають середовище процесу. Серед них об'єкти dentry, що відповідають кореневому каталогу процесу (поле root) і його поточному каталогу (поле pwd), а також права на доступ до файла за замовчуванням (поле umask).

Другим полем є files, що задає покажчик на структуру типу files_struct. Ця структура описує відкриті файли, із якими в цей час працює процес.

Таблиця файлових дескрипторів

Найважливішим полем структури files_struct є поле fd, що містить масив покажчиків на об'єкти file. Кожен елемент цього масиву відповідає файлу, відкритому процесом. Довжину масиву fd зберігають у полі max_fds.

Системні виклики UNIX використовують як параметр, що визначає відкритий файл, тобто індекс у масиві fd. Такі індекси називають *файловими дескрипторами* (file descriptors), а масив fd – масивом або таблицею файлових дескрипторів. Під час відкриття файла системний виклик open() виконує такі дії: створює новий об'єкт file; зберігає адресу цього об'єкта у першому вільному елементі масиву файлових дескрипторів fd; повертає індекс цього елемента (файловий дескриптор), який можна використати для роботи з цим файлом.

За замовчуванням під час створення процесу виділяють пам'ять під масив fd з 32 елементів. Якщо процес відкриє більше файлів, масив автоматично розширюється. Максимальна кількість відкритих файлів для процесу є одним із лімітів на ресурси для процесу, у Linux за замовчуванням вона дорівнює 1024, але може бути збільшена.

Перші три елементи масиву fd задаються автоматично під час запуску процесу, їм відповідають визначені файлові дескриптори, про які йтиметься у розділі 17.1.2.

Кілька елементів масиву fd можуть вказувати на один і той самий об'єкт file. Доступ через кожний із них призводить до роботи з одним і тим самим відкритим файлом. Є два основні способи домогтися такого дублювання дескрипторів.

По-перше, можна використати системні виклики dup() і dup2() для дублювання дескриптора в рамках одного процесу (dup2() опишемо в розділі 17).

По-друге, якщо створити процес-нащадок за допомогою fork(), елементи таблиці дескрипторів нащадка вказуватимуть на ті самі об'єкти file, що й відповідні елементи таблиці дескрипторів предка.

Розглянемо цю ситуацію докладніше. Насамперед з'ясуємо, чому необхідно вводити окремі об'єкти для відкритих файлів і список s_list замість того, щоб помістити інформацію про відкриті файли безпосередньо в таблицю дескрипторів файлів fd. Причина полягає в тому, що, якби інформація була поміщена безпосередньо в таблицю, предок і нащадок не могли б спільно використовувати відкриті файли так, як це потрібно для fork().

Прикладом може бути така послідовність подій.

1. Предок відкриває файл за допомогою open() і отримує дескриптор 4.
2. Предок записує три байти так: write(4,"AAA",3). Поточний покажчик позиції для дескриптора 4 переміщають у позицію 3, рахуючи від нуля.
3. Предок створює нащадка за допомогою fork() і очікує його завершення за waitpid().
4. Нащадок записує п'ять байтів: write(4,"BBBBB",5), переміщаючи поточний покажчик у позицію 8, після чого завершується.
5. Предок записує ще три байти: write(4,"CCC",3), після чого закриває файл.

У цьому разі коректна реалізація fork() вимагає, щоб предок продовжував записування у файл із того місця, на якому завершив своє виведення нащадок (тут — з позиції 8), тобто щоб у результаті було виведено "AAABBBBBCCC". У той же час, якби інформація про відкриті файли містилася в таблиці дескрипторів, предок після завершення нащадка не зміг би довідатися, що нащадок зробив поточну позицію рівною 8 (інформація про це була б вилучена із пам'яті разом із керуючим блоком нащадка після його завершення). У результаті предок продовжив би виведення з того місця, де він його завершив на кроці 2 (із позиції 3), поверх запису нащадка. Результат був би "AAACCCBB", що відповідно до POSIX невірно.

Взаємозв'язок компонентів VFS у разі отримання доступу до файла

Тепер є вся інформація для того, щоб описати і показати на рис. 13.2 взаємозв'язок між різними компонентами VFS, необхідними для отримання доступу до файла.

1. Керуючий блок процесу містить покажчик на масив файлових дескрипторів fd, системні виклики UNIX приймають як параметри індекси в цьому масиві. Кожен процес має окрему його копію.
2. Кожний елемент масиву файлових дескрипторів вказує на об'єкт відкритого файла (структуру file), ці об'єкти об'єднані у список. Кілька елементів таблиці fd (одного й того ж процесу або різних процесів) можуть вказувати на один і той самий об'єкт file.

3. Кожному об'єкту відкритого файла відповідає елемент каталогу, відображений об'єктом елемента каталогу (dentry). Кілька об'єктів file можуть посилатися на один об'єкт dentry; це означає, що файл був відкритий кілька разів із використанням одного й того самого імені (жорсткого зв'язку).
4. Кожному файлу файлової системи відповідає об'єкт індексного дескриптора (inode), ці об'єкти об'єднані у хеш-таблицю і декілька списків. Кілька об'єктів dentry можуть посилатися на один об'єкт inode, якщо вони відповідають жорстким зв'язкам.
5. Файловій системі в цілому відповідає об'єкт файлової системи. Інформація з об'єктів inode і об'єктів файлової системи дає змогу доступу до функцій підтримки конкретної файлової системи, що дає доступ до реального файла (на диску, мережного файла тощо).

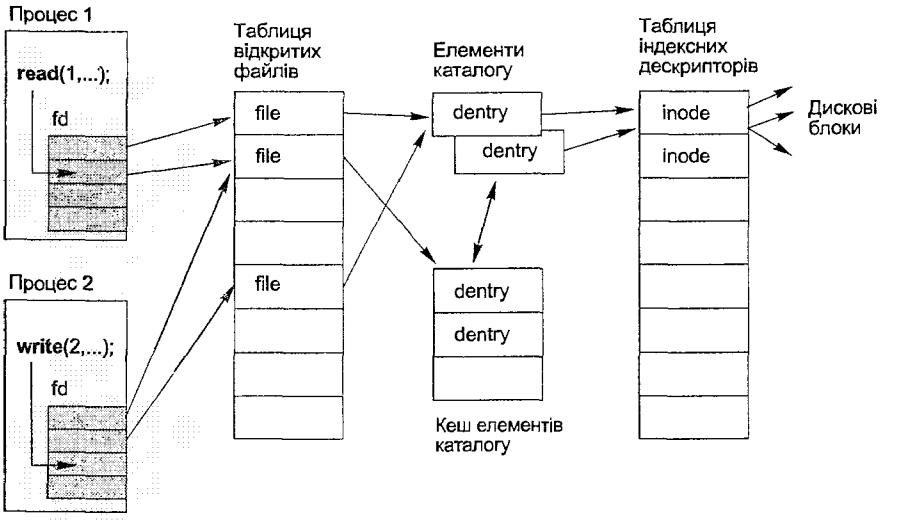

Рис. 13.2. Доступ до файла з процесу в Linux

13.2. Файлові системи ext2fs і ext3fs

Цей розділ присвячено файловим системам, які використовуються в Linux.

13.2.1. Файлова система ext2fs

Стандартною дисковою файловою системою для Linux є друга розширена файлова система (ext2fs). Розглянемо особливості її реалізації.

Ця файлова система багато в чому схожа на файлову систему FFS, про яку йшлося у розділі 12.3. Наприклад, спільними рисами є структура індексного дескриптора і розміщення каталогів на файловій системі як звичайних файлів (хоч їхній вміст інтерпретують трохи інакше). Спільною є також ідея групування індексних дескрипторів і блоків для зв'язаних даних, хоча для реалізації цієї ідеї використовують інший підхід.

Однією із базових відмінностей ext2fs від FFS є інша політика розподілу дискового простору. Як зазначалося, у FFS було дозволено розподіляти дисковий простір на блоки по 4 і 8 Кбайт, при цьому малі частини таких блоків, що залишилися після розподілу, своєю чергою розділяли на фрагменти по 1 Кбайт. В ext2fs ситуація змінилася – дисковий простір розподіляють на блоки тільки одного розміру. За замовчуванням він становить 1 Кбайт, хоча можна під час форматування файлової системи задати й більший розмір – 2 або 4 Кбайт.

Основою обробки запитів введення-виведення в ext2fs є кластеризація суміжних запитів для досягнення максимальної продуктивності, для чого, як і в FFS, прагнуть розташовувати зв'язані дані разом.

Для вирішення цієї задачі використовують групування даних, схоже за своїми принципами на використання груп циліндрів у FFS. Відмінності тут переважно зумовлені тим, що сьогодні у дисках використовують циліндри з різною геометрією залежно від відстані до центра пластини, тому об'єднання таких циліндрів у групи фіксованого розміру часто не відповідає фізичній структурі диска. Виходячи з цього, в ext2fs дисковий простір ділять не на групи циліндрів, а просто на групи блоків (block groups), не прив'язані до геометрії диска. Такі групи за структурою схожі на групи циліндрів: кожна група блоків теж є зменшеною копією файлової системи із суперблоком, таблицею індексних дескрипторів тощо (рис. 13.3).

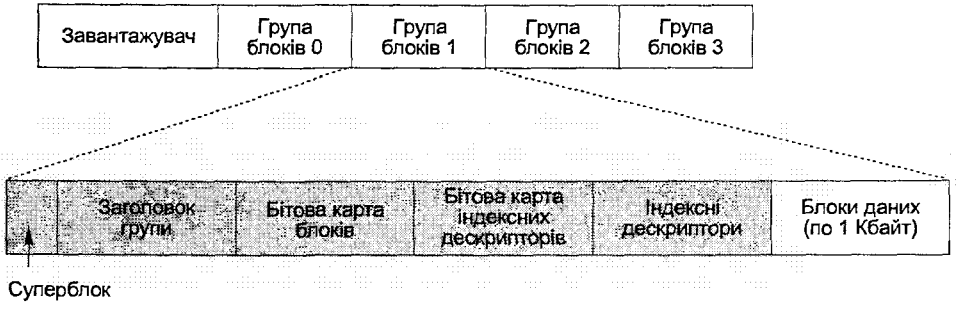

Рис. 13.3. Структура файлової системи ext2fs

Під час розміщення файла для нього спочатку вибирають групу блоків і в ній розміщують індексний дескриптор. При цьому перевагу отримує група блоків каталогу, де перебуває цей файл. Після цього, під час розподілу дискових блоків для файла, пробують вибрати ту саму групу, у якій перебуває індексний дескриптор. Файли каталогів не збирають разом, їх розподіляють рівномірно у доступних групах блоків для зменшення загальної фрагментації та більш рівномірного використання диска (навколо кожного файла каталогу групуватимуться індексні дескриптори його файлів, а навколо цих індексних дескрипторів – блоки їхніх файлів).

Облік вільних блоків та індексних дескрипторів ведуть за допомогою пари бітових карт – по одній на кожну групу блоків. Розмір кожної такої карти дорівнює одному блоку (1 Кбайт), тому максимально можлива кількість блоків та індексних дескрипторів у групі дорівнює 8 Кбайт. Під час розміщення перших блоків нового файла файлова система починає пошук вільних блоків від початку групи блоків, у разі розширення наявного файла пошук триває від блоку, виділеного найпізніше. Цей пошук відбувається у два етапи: на першому в бітовій карті відшукують

вільний байт (8 біт), що відповідає 8 неперервно розташованим дисковим блокам (аналогам блоку в 8 Кбайт). Якщо байт знайти не вдалося, виконують ще один пошук, але при цьому відшукують будь-який вільний біт. Якщо байтовий пошук завершився успішно, система шукає вільні біти від початку цього байта назад до першого зайнятого біта. Метою цього пошуку є зменшення фрагментації (так не залишатиметься «дірок» між виділеним інтервалом блоків і попередніми зайнятими блоками).

Після того, як вільний блок було знайдено одним із двох способів, його виділяють для використання файлом. Додатково випереджувально виділяють до 8 блоків, що розташовуються за ним (якщо зайнятий блок під час цього виділення виявляють раніше, то виділяють менше ніж 8 блоків). Метою такого випереджувального виділення є зменшення фрагментації та витрат часу на виділення дискових блоків. У разі закриття файла всі виділені таким чином блоки, що реально не використані, повертаються у бітову карту як вільні.

Розмір кожного індексного дескриптора становить 128 байт. Використовують 12 прямих блоків і по одному непрямому блоку першого, другого та третього рівнів. Довжина адреси блоку становить 4 байти, що більше, ніж стандартна довжина для багатьох UNIX-систем, тому можна адресувати більше дискового простору. Місце для розміщення списків керування доступом зарезервоване, але може бути використане тільки в ядрі версії 2.6.

13.2.2. Особливості файлової системи ext3fs

Файлова система ext3fs являє собою розширення ext2fs внаслідок додання журналу. Важливою її особливістю є можливість відображення в журналі змін не тільки метаданих файлової системи, але й файлових даних (підтримку цієї можливості задають під час монтування системи). Це призводить до істотного зниження продуктивності, але дозволяє підвищити надійність.

Можуть бути задані три режими роботи із журналом: *режим журналу* (journal), за якого всі зміни даних зберігаються в журналі; *упорядкований режим* (ordered), за якого зберігаються тільки зміни в метаданих, але при цьому файлові операції групують так, щоб блоки даних зберігалися на диску перед метаданими, внаслідок чого знижується небезпека ушкодження інформації всередині файлів; *режим мінімального записування* (writeback), за якого зберігаються тільки метадані.

За замовчуванням використовують упорядкований режим.

Журнал файлової системи ext3fs зберігають у схованому файлі .journal у кореневому каталозі файлової системи. Її код не працює із файлом журналу безпосередньо, для цього використовують спеціальний рівень коду ядра із назвою JBD (Journaling Block Device Layer). Код JBD групує дискові операції в транзакції, інформацію про які фіксують у журналі. Система гарантує, що інформація про підтверджену транзакцію буде вилучена із журналу лише тоді, коли всі відповідні блоки даних записані на диск.

Під час завантаження системи після збою утиліта e2fsck переглядає журнал і планує до виконання всі операції записування, описані підтвердженими транзакціями.

13.3. Файлова система /proc

Принципи реалізації

Найцікавішим прикладом реалізації інтерфейсу файлової системи VFS для доступу до даних, що не перебувають на диску, є файлова система /proc. Ці дані насправді не зберігаються ніде, вміст кожного файла і каталогу генерують «на льоту» у відповідь на запити користувача.

Така файлова система вперше з'явилася в UNIX System V Release 4. Вона ґрунтувалась на тому, що кожному процесові у системі відповідає каталог файлової системи /proc, при цьому ім'я каталогу має збігатися з цифровим зображенням ідентифікатора (pid) цього процесу (наприклад, процесу із pid=25 має відповідати каталог /proc/25). Із каталогу можна дістати доступ до різних файлів із визначеними іменами, при цьому доступ до кожного з них має спричиняти генерування різної інформації про процес у текстовому форматі, зручному для синтаксичного аналізу. Такою інформацією може бути вміст командного рядка процесу – /proc/pid/cmdinfo, відомості про поточне завантаження ним процесорів – /proc/pid/cpu, різноманітна статистика – /proc/pid/status тощо. В цілому вся інформація, відображувана програмою ps, має бути доступна через дану файлову систему. Ця інформація є динамічною. Коли, наприклад, переглянути двічі поспіль файл, що відображає поточне завантаження процесора, можемо отримати різні результати.

У Linux реалізовано вищеописане подання інформації про процеси, але, крім цього, у відображення /proc було додано багато нових файлів і каталогів. Основним призначенням цих засобів доступу є надання різної статистики щодо системи, зокрема частина цих файлів і каталогів надає інтерфейс до служб ядра. За допомогою цієї системи можна здобути вичерпну інформацію про завантажені модулі ядра (/proc/modules), змонтовані файлові системи (/proc/mounts), зовнішні пристрої (/proc/devices, /proc/pci, /proc/ide), процесори (/proc/cpuinfo), стан пам'яті (/proc/meminfo) тощо.

Особливо важливий інтерфейс, що забезпечує доступ до внутрішніх змінних ядра; він реалізований через файли в каталогах /proc/sys і /proc/sys/kernel. Суттєвим тут є той факт, що /proc підтримує не лише читання значень таких змінних, але їхнє редагування без перезавантаження системи і перекомпіляції ядра (через записування нових значень у відповідні файли). Прикладом змінної, котра може бути відредагована через інтерфейс /proc, є максимально допустима кількість потоків у системі, що може бути модифікована під час її роботи шляхом зміни файла /proc/sys/kernel/threads-max:

```
# echo 10000 > /proc/sys/kernel/threads-max
```

Реалізація цієї файлової системи ґрунтується на тому, що кожному файлу і каталогу в ній присвоюють унікальний і незмінний номер індексного дескриптора. У разі доступу до файла цей номер передають у відповідний метод об'єкта індексного дескриптора VFS (наприклад, метод читання файла). Цей метод замість звертання до диска просто перевіряє номер індексного дескриптора і залежно від його значення виконує потрібний код (наприклад, зчитує інформацію з керуючого блока відповідного процесу).

Використання файлової системи /proc із прикладних програм

Використання файлової системи /proc із прикладних програм організоване дуже просто [24]. Необхідно виконувати стандартні системні виклики роботи з файлами,

шлях до яких відомий, зчитувати із них інформацію у текстовому форматі, виконувати її синтаксичний розбір і виділяти потрібні дані. Жодних додаткових прав доступу для цього не потрібно. Відомий приклад того, як змінилася реалізація утиліти ps (що відображає інформацію про процеси у системі) із появою /proc. Якщо раніше вона була реалізована як привілейований процес, котрий зчитував інформацію про процеси безпосередньо з пам'яті ядра, то тепер її стало можливо реалізувати як звичайну прикладну програму, що зчитує і форматує текстові дані, доступні через /proc.

Прикладом використання файлової системи /proc може бути визначення загального обсягу пам'яті в системі. Цю інформацію повертають після читання із файла /proc/meminfo. Перші два рядки його мають такий вигляд:

```
        total:    used:    free:  shared: buffers:  cached:
Mem:  63754240 60063744  3690496    49152  2600960 31346688
```

Тут видно, що потрібна нам інформація перебуває у другому рядку. Програма має відкрити файл, зчитати з нього дані та знайти в них потрібний фрагмент:

```
int fd1, bytes_read, total_mem;
char meminfo[1024], *match;
fd1 = open("/proc/meminfo", O_RDONLY);          // відкриття /proc/meminfo
bytes_read = read(fd1, meminfo, sizeof(buf));   // читання даних
meminfo[bytes_read] = '\0';
match = strstr(meminfo, "Mem:");                // пошук фрагмента
sscanf(match, "Mem: %d", &total_mem);           // зчитування інформації
printf("Усього пам'яті: %d\n", total_mem);
close(fd1);
```

13.4. Файлові системи лінії FAT

Згадуючи про файлові системи лінії FAT, матимемо на увазі кілька близьких за організацією файлових систем, які розрізняють за способом адресації кластера на диску (FAT-12, FAT-16, FAT-32) або наявністю підтримки довгих імен (така підтримка є для всіх реалізацій FAT, використовуваних у Consumer Windows і в лінії Windows XP). У цьому розділі вивчатимемо тільки системи із підтримкою довгих імен, а на відмінностях в адресації зупинимося окремо. Якщо виклад не зачіпатиме відмінностей в адресації, вживатимемо назву *FAT*, розуміючи під нею кожну із файлових систем цього сімейства.

Структура розділу FAT

Розглянемо структуру розділу, що містить файлову систему FAT. Вона відповідає базовій структурі, описаній у розділі 12.2.4.

- Після завантажувального сектора, в якому знаходиться завантажувач системи, розташовані дві копії таблиці розміщення файлів (FAT). Резервну копію FAT використовують для відновлення основної копії у разі її ушкодження. Усі операції, що ведуть до зміни FAT, негайно синхронізують із резервною копією.
- Далі розташований кореневий каталог, під який виділяють 32 Кбайт, що дає змогу зберігати в ньому 512 елементів (на кожний елемент виділено 32 байта).
- За кореневим каталогом слідує ділянка даних, у якій розташовані всі файли і каталоги, крім кореневого.

Про структуру FAT ідеться у розділі 12.2.4, її елементи називають індексними покажчиками. Під час розміщення файла FAT проглядають від початку в пошуках першого вільного індексного покажчика. Після цього в поле каталогу, що відповідає номеру першого кластера, заносять номер цього покажчика. Далі будують ланцюжок покажчиків у FAT. В останній індексний покажчик файла заносять ознаку кінця файла. Зазначимо, що якщо кластери, які слідують за початковим кластером файла, у момент розміщення виявляться вільними (це найчастіше буває після форматування розділу), файл займе суміжні кластери і буде неперервним, у противному разі між кластерами цього файла на диску розташовуватимуться кластери інших файлів. Така ситуація відповідає зовнішній фрагментації дискового простору.

Особливості адресації FAT

Тепер зупинимося на відмінностях у версіях FAT, що ґрунтуються на особливостях адресації. Найважливішою характеристикою FAT є *розмір індексного покажчика*. Оскільки кожен із покажчиків вказує на кластер, від їхнього розміру залежить загальна кількість кластерів на диску і розмір FAT. Відмінності між версіями FAT визначаються розміром індексного покажчика: для FAT-12 він дорівнює 12 біт (що відповідає 4 Кбайт кластерів), для FAT-16 — 16 біт (64 Кбайт кластерів), для FAT-32 — 32 біти (2^{32} кластери).

Максимальний обсяг розділу, що може бути адресований FAT конкретної версії, залежить від розміру кластера і максимальної кількості адресованих кластерів, обумовленої довжиною індексного покажчика. Що більший кластер, то менше їх потрібно для адресації того самого обсягу диска і навпаки; з іншого боку, великий розмір кластера спричиняє значну внутрішню фрагментацію.

Звичайно вибирають мінімальний розмір кластера, який дає змогу адресувати весь розділ визначеного обсягу, при цьому бажано, щоб кластер не був більший за 4 Кбайт. Наприклад, для FAT-12 і розміру кластера 4 Кбайт обсяг розділу не може перевищувати 16 Мбайт (тому таку систему рекомендують лише для дискет). Система FAT-16 за такого розміру кластера може бути застосована для розділів до 512 Мбайт, для більших розділів потрібно збільшувати розмір кластера. Наприклад, для розділу розміром понад 1 Гбайт розмір кластера має бути 32 Кбайт. Із подоланням цього обмеження насамперед пов'язане впровадження FAT-32, що дає змогу використати кластери на 4 Кбайт для розділів розміром до 8 Гбайт.

Розмір FAT залежить від розміру індексного покажчика та обсягу розділу, у FAT-32 для великих розділів вона може досягати кількох мегабайт. ОС звичайно кешує FAT, але якщо зовнішня фрагментація диска значна і розмір FAT великий, ефективне кешування може бути ускладнене, внаслідок чого знижується продуктивність системи.

Структура елемента каталогу

Елемент каталогу в FAT містить: ім'я файла у форматі 8.3 (імена файлів розглянемо пізніше); поле атрибутів (1 байт) — тільки для читання, системний, схований; дату і час останньої модифікації файла; дату останнього доступу; номер першого кластера файла (4 байти); розмір файла (4 байти).

Особливості вилучення файлів

У разі вилучення файла перший символ відповідного елемента каталогу заміняють на символ 'x', який означає, що цей елемент можна використовувати заново, а всі кластери файла у FAT позначають як вільні. Номер першого кластера і довжину файла в елементі каталогу не знищують, що дає можливість відновити вміст вилученого файла, коли його кластери розташовані послідовно (на цьому засновано дія утиліт відновлення файлів після вилучення).

Зберігання довгих імен

До появи Windows 95 FAT надавала змогу використовувати тільки імена, що складаються з 11 значущих символів (8 – ім'я файла, 3 – розширення), при цьому вони зберігалися у відповідному елементі каталогу. Введення довгих імен призвело до того, що на додачу до традиційного імені (яке тепер називають «коротким іменем») у таких записах каталогу зберігають ім'я завдовжки до 255 символів фрагментами по 13 двобайтових Unicode-символів. При цьому коротке ім'я отримують із довгого шляхом додавання до перших 6 символів у верхньому регістрі суфікса ~1, ~2 і т. д., залежно від наявності таких імен до цього часу в каталозі.

Для того щоб відрізнити фрагмент довгого імені від елемента каталогу, котрий відповідає файлу, для записів із фрагментами імені задають значення атрибутів 0x0F, що як атрибут файла не може бути використане. Для відстеження відповідності між елементом каталогу і довгим іменем використовують поле контрольної суми.

13.5. Файлова система NTFS

Файлова система NTFS [14, 44, 69, 70] є основною файловою системою ОС лінії Windows XP. Головними її перевагами є надійність, високий ступінь захищеності та раціональне використання дискового простору.

Про засоби захисту даних, реалізованих у NTFS, ітиметься в розділі 18.

13.5.1. Розміщення інформації на диску

Логічний розділ із розміщеною файловою системою у NTFS називають *томом*. Усі метадані (включаючи інформацію про структуру тому) зберігають у файлах на диску. Про ці файли докладніше розповімо нижче.

Розмір кластера задають під час високорівневого форматування розділу диска, і він може варіюватися від розміру сектора диска до 4 Кбайт. Кожному кластеру присвоюють логічний номер (Logical Cluster Number, LCN), який використовують у системі для ідентифікації кластера. Фізичний зсув на диску в цьому разі може бути отриманий множенням розміру кластера на LCN.

Загальна структура тому NTFS показана на рис. 13.4.

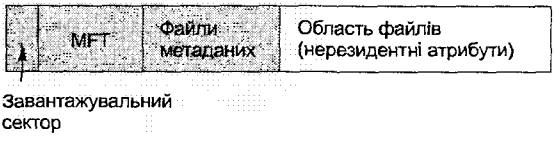

Рис. 13.4. Структура тому NTFS

Відображення файлів

Файл для NTFS відображається складніше, ніж для файлових систем UNIX та сімейства FAT. Він складається із набору атрибутів, причому кожний з них є незалежним *потоком* (stream) байтів, який можна створювати, вилучати, зчитувати і записувати. Деякі атрибути є стандартними для всіх файлів, до них належать ім'я (FileName), версія (Version), стандартна інформація про файл, що включає час створення, час відновлення тощо (Standard Information). Інші атрибути залежать від призначення файла (наприклад, каталог включає як атрибут Index Root структуру даних, що містить список файлів цього каталогу). Є універсальний атрибут Data, що містить всі дані файла (*потік даних*). Для звичайних файлів він може бути єдиним із необов'язкових.

Важливою характеристикою NTFS є те, що файл може містити більш як один атрибутів Data, які розрізняють за іменами. У зв'язку з цим кажуть, що дозволена наявність кількох поіменованих потоків даних усередині файла [69].

За замовчуванням файл містить один потік даних, що не має імені. Для доступу до додаткових потоків використовують звичайні функції роботи із файлами (`CreateFile()`, `WriteFile()`, `ReadFile()` тощо), ім'я потоку при цьому записують через двокрапку після імені файла:

```
DWORD bytes_read, bytes_written;
// створення нового поіменованого потоку у файлі
HANDLE hf = CreateFile("myfile.txt:mystream",
     GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_NEW, 0, 0);
// записування у цей потік
WriteFile(hf, "hello\n", 7, &bytes_written, 0);
CloseHandle(hf);
```

Зазначимо, що під час звертання до такого файла за іменем (`myfile.txt`) отримається доступ тільки до стандартного безіменного потоку. З іншого боку, у разі спроби перенести такий файл у файлову систему, що не підтримує кілька потоків усередині файла (наприклад, FAT), буде видано попередження про можливу втрату даних.

Головна таблиця файлів (MFT)

Кожний файл у NTFS описують одним або кількома записами в масиві, що зберігається у спеціальному файлі — *головній таблиці файлів* (Master File Table, MFT). Розмір такого запису залежить від розміру розділу і звичайно становить 2 Кбайт. Атрибути невеликого розміру (наприклад, ім'я файла), які зберігають у записі MFT безпосередньо, називаються *резидентними атрибутами*. Атрибути великого обсягу (наприклад, ті, що містять дані файла), зберігають на диску окремо — це *нерезидентні атрибути*. Розміщують такі дані екстентами (групами кластерів заданої довжини), при цьому покажчик на кожен екстент зберігають у записі MFT (такий покажчик складається із трьох елементів: порядкового номера кластера на томі (LCN), номера кластера всередині файла (VCN) і довжини екстента). Максимальний розмір MFT становить 245 записів.

Можна виділити кілька способів зберігання файлів залежно від їхнього розміру.

- Файли малого розміру можуть бути розміщені повністю в одному записі MFT і виділення додаткових екстентів не потребують.

- Файли більшого розміру потребують використання нерезидентних атрибутів зі зберіганням покажчиків на кожен екстент такого атрибута в атрибуті Data вихідного запису MFT.
- Файли із великим набором атрибутів або високою фрагментацією (коли атрибут Data не вміщує всіх покажчиків на екстенти) можуть потребувати для розміщення кілька записів MFT. Основний запис у цьому разі ще називають базовим записом файла (base file record), інші — записами переповнення (overflow records). Базовий запис файла містить список номерів записів переповнення в атрибуті Attribute List. Приклад такого зберігання інформації показано на рис. 13.5.
- У разі файлів надзвичайно великого розміру в базовому записі може бракувати місця для зберігання списку номерів записів переповнення, тому атрибут Attribute List можна зробити нерезидентним.

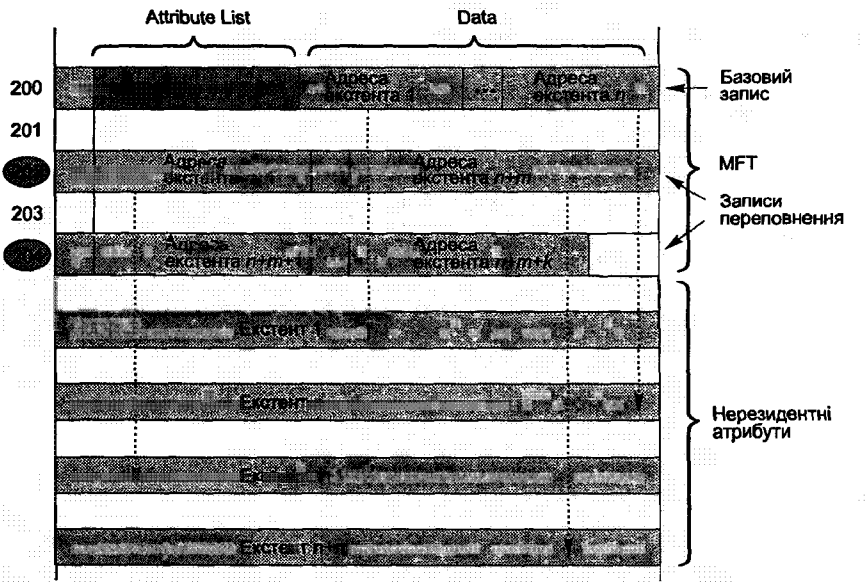

Рис. 13.5. Зберігання файла в кількох записах MFT

Кожен файл на томі NTFS має унікальний ідентифікатор — файлове посилання (file reference). Розмір такого посилання становить 64 біти, з них 48 біт займає номер файла (індекс у MFT), а 16 біт — номер послідовності, який збільшують під час кожного нового задання елемента MFT і використовують для контролю несуперечливості файлової системи.

Ідентифікація файла за його номером у MFT подібна до індексованого розміщення файлів, при цьому MFT відіграє роль ділянки індексних дескрипторів. Однак, на відміну від індексних дескрипторів, записи MFT можуть безпосередньо містити дані файла.

Відображення каталогів

Каталоги у NTFS зберігають двома способами. Для невеликих каталогів зберігають лише список їхніх файлів, використовуючи для цього резидентний атрибут

Index Root. Коли каталог досягає певного розміру, з ним пов'язують спеціальну структуру даних — B+-дерево, основною властивістю якого є те, що шлях від кореня до будь-якого вузла завжди однієї довжини. Кожен елемент такого дерева містить відповідне файлове посилання, а також копії деяких атрибутів MFT (серед них ім'я файла, час доступу, розмір файла). Це зроблено для прискорення виконання операцій, які не вимагають доступу до даних файла, наприклад перелічення елементів каталогу.

Файли метаданих

Як зазначалося, усі метадані тому зберігають у спеціальних файлах. Першим із них ($Mft) є MFT (тобто як перший запис MFT містить посилання на себе), другий ($MftMirr) містить резервну копію перших 16 записів MFT. Розглянемо деякі інші спеціальні файли.

- Файл журналу ($LogFile) — містить інформацію про журнал файлової системи. Про структуру цього файла йтиметься в розділі 13.5.3.
- Файл тому ($Volume) — містить ім'я тому, версію NTFS і біт, вмикання якого означає, що том, можливо, був ушкоджений і його потрібно перевірити під час завантаження.
- Таблиця визначення атрибутів ($AttrDef) — задає набір атрибутів тому і допустимих операцій над ними.
- Файл \ — відображає кореневий каталог.
- Файл бітової карти ($BitMap) — містить бітову карту кластерів диска з індикацією зайнятих кластерів.
- Файл $Boot — відображає завантажувальний сектор, розташований за адресою, де його може знайти завантажувач BIOS (процес завантаження Windows XP буде описано у розділі 19). У ньому також зберігають адрес MFT.
- Файл зіпсованих кластерів ($BadClus) — відстежує зіпсовані кластери на диску. Система NTFS використовує технологію перерозподілу збійних кластерів, що дає змогу підставляти під час звертання до збійних кластерів дані коректних, приховуючи цим наявність проблем.
- Файл $Security — містить загальні атрибути безпеки.

Точки повторного аналізу

Починаючи із Windows 2000, у NTFS є можливість задавати *точки повторного аналізу* (reparse points) [70]. Така точка — це спеціальний файл, пов'язаний із блоком даних, що містить виконуваний код. Коли під час аналізу шляху до файла трапляється така точка, відбувається програмне переривання, що передає керування коду пов'язаного з нею блоку. Він зазвичай переносить подальший пошук на інший каталог або інший том — так можуть бути реалізовані зв'язки і монтування файлових систем.

Для монтування файлової системи із прикладної програми використовують функцію SetVolumeMountPoint():

```
BOOL SetVolumeMountPoint(LPCTSTR mount_point, LPCTSTR volume);
```

Тут mount_point — ім'я наявного каталогу (включаючи завершальний зворотний слеш), що стане точкою монтування, наприклад "c:\\disk1\\"; volume — унікальне

ім'я, що ідентифікує том із файловою системою і має формат \\?\Volume{GUID}, де GUID є унікальним 128-бітовим ідентифікатором, який широко застосовують у Windows-системах.

Для того щоб отримати унікальне ім'я тому за його символьним іменем (C:\ тощо), потрібно виконати такий код:

```
char volname[1024];   // унікальне ім'я тому
GetVolumeNameForVolumeMountPoint("C:\\", volname, sizeof(volname));
```

Тепер код для монтування матиме такий вигляд:

```
SetVolumeMountPoint("D:\\disk1\\", volname);
```

Для розмонтування потрібно використати функцію DeleteVolumeMountPoint():

```
DeleteVolumeMountPoint("D:\\disk1\\");
```

13.5.2. Стискання даних

Засоби стискання даних

Система NTFS може робити стискання як окремих файлів, так і всіх файлів у каталозі [44, 70]. Для цього дані файла розділяють на одиниці стискання, що є групами із 16 суміжних кластерів. Під час записування на диск кожної такої групи до неї застосовують алгоритм стискання. Якщо при його застосуванні було досягнуто виграш у дисковому просторі, після даних стиснутих 16 кластерів міститься посилання на екстент із нульовою адресою, яке означає, що попередні 16 кластерів були стиснуті. Якщо стиснути групу кластерів не вдалося, її зберігають на диску без змін. Коли необхідно прочитати файл, NTFS визначає всі стиснуті групи (після яких є посилання на екстенти із нульовими адресами), дані цих екстентів зчитують і розпаковують у випереджувальному режимі. Зауважемо, що для довільного доступу до стиснутого файлу необхідно розпакувати всі дані від початку файла до позиції доступу, вибір 16 кластерів як одиниці стискання є спробою досягти в цьому разі компромісу між швидкістю доступу та ефективністю стискання.

Функція GetFileSize() для таких файлів повертає розмір файлу без урахування стискання. Для того щоб дізнатися, скільки місця на диску займає стиснутий файл (тобто його розмір після стискання), необхідно скористатися функцією GetCompressedFileSize():

```
HANDLE fh = CreateFile("myfile.txt", GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL);
DWORD size = GetFileSize(fh, NULL);
DWORD size_on_disk = GetCompressedFileSize("myfile.txt", NULL);
printf("розмір без стискання: %d, зі стисканням: %d\n", size, size_on_disk);
```

Підтримка розріджених файлів

Починаючи із версії для Windows 2000, у NTFS з'явилася підтримка розріджених файлів (див. розділ 12.2.5). Для таких файлів задається спеціальний атрибут, після чого в послідовності номерів кластерів елемента MFT можуть бути утворені «діри», якщо після створення файла до цих кластерів не було звертання. У подальшому, якщо під час читання файла трапляється така «діра», замість неї без

доступу до диска система повертає блок, заповнений нулями. Зазначимо, що можна явно задавати ділянки файла, до яких не планують звертатися.

Для задання атрибутів стискання і розрідженості необхідно використати низькорівневу функцію керування введенням-виведенням `DeviceIoControl()`, яку розглянемо в розділі 15 [70].

13.5.3. Забезпечення надійності

Основою забезпечення надійності у NTFS є те, що це — журнальна файлова система (про принципи роботи таких систем ішлося у розділі 12). Усі зміни структур даних файлової системи відбуваються всередині атомарних транзакцій, які виконуються повністю або не виконуються зовсім. У журнал записують інформацію про початок і підтвердження транзакції. У разі відновлення після збою спочатку повторно виконують дії всіх підтверджених транзакцій, а потім відміняють дії всіх тих, які не були підтверджені (у журналі транзакцій завжди наявна інформація, необхідна для виконання як однієї, так і іншої дії).

Кожні 5 с у журналі зберігають інформацію про точку перевірки, тоді ж інформацію про зміни файлів зберігають на диску остаточно. Записи журналу транзакцій до точки перевірки не потрібні й можуть бути вилучені.

Журнал зберігають у файлі метаданих із назвою `$LogFile`, який створюють під час форматування розділу. Він складається з двох секцій: ділянки записів журналу (logging area), яка являє собою циклічний список записів журналу, і ділянки перезапуску (restart area), що містить дві ідентичні копії поточної інформації про стан журналу (наприклад, там зберігають позицію, з якої потрібно буде почати відновлення).

Якщо в журналі бракує місця, Windows XP ставить транзакції в чергу, і всі нові операції введення-виведення відміняються. Коли всі поточні операції виконані, NTFS звертається до менеджера кеша, щоб той записав весь кеш на диск, після чого журнал очищають і виконують усі відкладені транзакції.

13.6. Особливості кешування у Windows XP

У Windows XP реалізовано єдиний кеш, що обслуговує всі файлові системи. Керування цим кешем здійснює *менеджер кеша* (Cache Manager).

Основною особливістю менеджера кеша є те, що він працює на більш високому рівні, ніж файлові системи (на відміну, наприклад, від традиційної архітектури UNIX, де кеш розташований нижче від файлових систем).

Менеджер кеша тісно взаємодіє із менеджером віртуальної пам'яті. Розмір кеша міняють динамічно залежно від обсягу доступної пам'яті. Менеджер віртуальної пам'яті резервує для системного кеша половину системної ділянки адресного простору процесу (верхні 2 Гбайт). У своїй роботі менеджер кеша використовує технологію відображуваної пам'яті, як описано у розділі 12.3.2: файли відображаються на цей адресний простір, після чого відповідальність за керування введенням-виведенням передають менеджерові віртуальної пам'яті. Кеш розділяють на блоки по 256 Кбайт, кожен із блоків може відображати ділянку файла.

Блоки у кеші описують за допомогою керуючого блоку віртуальної адреси або блоку адреси (Virtual Address Control Block, VACB), що містить віртуальну адресу

файла і зсув усередині файла для цього блоку. Глобальний список таких блоків підтримує менеджер кеша.

Для відкритих файлів підтримують окремий масив покажчиків на блоки адреси. Кожен елемент масиву відповідає ділянці файла розміром 256 Кбайт і вказує на блок адреси, якщо ділянка файла перебуває в кеші, у протилежному випадку він містить нульовий покажчик.

У разі спроби доступу до файла менеджер кеша визначає за зсувом, якому блоку адреси відповідає запит. Якщо відповідний елемент масиву нульовий, відсилають запит драйверу файлової системи на читання відповідного фрагмента файла із диска в кеш, після чого копіюють дані з кеша у буфер застосування користувача.

Для підвищення продуктивності менеджер кеша відстежує історію трьох попередніх запитів, і коли він може знайти в них закономірність, буде виконано випереджувальне читання на підставі цієї закономірності (наприклад, якщо задають послідовність читання файла, випереджувальне читання зберігатиме в кеші додаткові блоки, розташовані у напрямку читання). Обсяг випереджувального читання фіксований і становить 192 Кбайт.

Зазначимо, що файл завжди відображають у пам'ять тільки один раз незалежно від того, скільки процесів до нього звертаються. При цьому сторінки із кеша копіюватимуться у буфер кожного процесу, так може бути забезпечена несуперечливість відображення файла для різних процесів.

13.7. Системний реєстр Windows XP

Цей розділ присвячено опису найважливішого компонента Windows XP — системного реєстру (registry) [16, 44]. *Реєстр* — це ієрархічно організоване сховище інформації про налаштування системи і прикладних програм. Крім цього, реєстр використовують для перегляду даних про поточний стан системи.

Незважаючи на те що на фізичному рівні реєстр не є файловою системою, його доцільно розглянути в цьому розділі, оскільки на логічному рівні він дуже подібний до неї. Крім того, реєстр зберігають у файлах на диску.

Важливість реєстру зумовлена тим, що в ньому міститься інформація, необхідна для завантаження і функціонування системи. Втрата або некоректна зміна даних реєстру можуть спричинити непрацездатність системи.

13.7.1. Логічна структура реєстру

На логічному рівні реєстр можна розглядати як ієрархічну файлову систему із кількома кореневими каталогами. Аналогом каталогів у цьому разі є *ключі* (keys), аналогом файлів — *значення* (values). Ключі характеризуються іменами і містять значення або інші ключі. Кожне значення характеризується іменем, типом і даними, які воно містить. Найпоширенішими типами значень є REG_SZ — текстовий рядок, REG_DWORD — ціле число розміром 4 байти, REG_BINARY — двійкові дані довільної довжини. Крім цього, можливі посилання на інші значення або ключі (ці посилання аналогічні до символічних зв'язків файлових систем).

Як і у файловій системі, кожне значення характеризується повним шляхом, що включає всі імена ключів, розташованих над ним.

Розглянемо кореневі каталоги реєстру (ключі верхнього рівня). Найважливішими з них є HKEY_LOCAL_MACHINE (скорочено HKLM) і HKEY_USERS (HKU). Саме ці ключі відповідають фізичним даним реєстру. Ключ HKLM містить інформацію про всю систему, HKU – дані окремих користувачів.

Підмножину дерева ключів, починаючи із ключа другого рівня, називають вуликом (hive). Під ключем HKLM розташований ряд важливих вуликів:

- HARDWARE – містить інформацію про поточну апаратну конфігурацію системи; його вміст формують динамічно і на диску не зберігають;
- SAM – база даних облікових записів, містить інформацію про імена і паролі користувачів, необхідну для реєстрації у системі (ця інформація розглядатиметься в розділі 18);
- SOFTWARE – зберігає налаштування прикладного програмного забезпечення (звичайно підключі цього вулика називають за іменем фірми-виробника);
- SYSTEM – містить інформацію, необхідну під час запуску системи, зокрема список драйверів і служб, які необхідно завантажити, а також їхні налаштування.

В реєстрі можуть зберігатися різні значення.

- Прикладом загальносистемного налагодження є значення HKLM\SYSTEM\CurrentControlSet\Services\Cdrom\Autorun типу REG_DWORD, що може містити 0 або 1. Коли воно містить 1, вставлення нового диска у CD-дисковід приводить до автоматичного запуску застосування autorun.exe, якщо воно є на цьому диску.
- Прикладом засобу зберігання налагодження програмного продукту може бути ключ HKLM\SOFTWARE\Adobe\Acrobat Reader\6.0, що містить значення конфігураційних параметрів цієї версії продукту.

Ключ HKU містить профілі користувачів (налаштування їхнього робочого стола, конфігурацію застосувань тощо). Інформацію щодо кожного користувача зберігають у вулику, ім'я якого збігається з ідентифікатором безпеки (SID) цього користувача. Про SID ітиметься в розділі 18.

Інші ключі верхнього рівня відображають динамічні дані або посилання на інші ключі. Наприклад, ключ HKEY_PERFORMANCE_DATA є засобом доступу до різних поточних характеристик ОС, подібно до файлової системи /proc. А ключ HKEY_CURRENT_USER це посилання на ключ HKU\SID-поточного_користувача і відповідає налаштуванням поточного користувача.

13.7.2. Фізична організація реєстру

Більша частина реєстру зберігається на диску у файлах, що відповідають вуликам (*файлах вуликів* – hive files). Файли вуликів HKLM розташовані в підкаталозі System32\Config системного каталогу Windows. Імена цих файлів збігаються з іменами вуликів (System, Software тощо). Вулики налаштувань користувачів (HKU\SID) зберігаються як файли Documents And Settings\ім'я_користувача\NTUSER.DAT.

Зміни файлів вуликів відбуваються за тими самими правилами, що і для журнальних файлових систем (на базі атомарних транзакцій), крім того, для вулика System автоматично підтримують резервну копію.

13.7.3. Програмний інтерфейс доступу до реєстру

Win32 API надає функції, що дозволяють виконувати різні дії з реєстром. Подивимося, як за їхньою допомогою можна зчитувати інформацію з реєстру, створювати нові ключі та значення.

Для читання інформації з реєстру необхідно насамперед відкрити ключ, у якому перебуває потрібне значення. Для цього використовують функцію RegOpenKeyEx():

```
HKEY hk;
RegOpenKeyEx(HKEY_LOCAL_MACHINE,  // HKEY_CURRENT_USER тощо
"SYSTEM\CurrentControlSet\Services\Cdrom", 0, KEY_READ, &hk);
```

Останнім параметром ця функція приймає покажчик на змінну, в яку буде записано дескриптор ключа реєстру. Після цього необхідно отримати дані потрібного значення за допомогою функції RegQueryValueEx(), куди передають такий відкритий дескриптор:

```
DWORD vsize, autorun;
// RegOpenKeyEx(..., &hk);
RegQueryValueEx(hk, "Autorun", NULL, NULL, (LPBYTE)&autorun, &vsize);
// autorun містить 0 або 1
```

Після роботи із ключем потрібно його закрити за допомогою функції

```
RegCloseKey(hk);
```

Для створення нового ключа використовують функцію RegCreateKeyEx(), а для створення нового значення всередині ключа — RegSetValueEx(). Наведемо приклад їхнього використання:

```
char myval[] = "my new data";
HKEY hknew;
RegCreateKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\myapp", 0, NULL, 0,
0, NULL, &hknew, &res);
RegSetValueEx(hknew, "myval", 0, REG_SZ, (LPBYTE)myval, sizeof(myval));
RegCloseKey(hk);
```

Висновки

- Важливою концепцією доступу до файлової системи, реалізованою в UNIX-системах, зокрема в Linux, є абстрагування інтерфейсу різних файлових систем від прикладних програм за допомогою програмного забезпечення віртуальної файлової системи (VFS). VFS дає змогу в разі використання різних файлових систем обмежитися базовим набором операцій доступу. На основі VFS можна реалізовувати доступ через інтерфейс файлової системи до даних, які за своєю природою з дисковими файлами не пов'язані.
- Основною файловою системою, яку використовують у Linux, є ext2fs. За структурою вона подібна до системи FFS.
- В ОС лінії Windows XP здебільшого використовують NTFS. Ця система підтримує журнал, а також шифрування і стискання даних.

Контрольні запитання та завдання

1. Опишіть можливу реалізацію системних викликів `open()`, `read()` і `close()` у Linux з урахуванням інтерфейсу VFS і наявності дискового кеша.
2. Поясніть, коли використання реалізації VFS для Linux призведе до того, що:
   а) два файлові дескриптори посилатимуться на один файловий об'єкт;
   б) один файловий дескриптор посилатиметься на два файлових об'єкти;
   в) два файлові об'єкти посилатимуться на один об'єкт елемента каталогу;
   г) один файловий об'єкт буде відповідати двом елементам каталогу;
   д) два елементи каталогу посилатимуться на один об'єкт індексного дескриптора;
   е) один об'єкт каталогу відповідатиме двом об'єктам індексного дескриптора;
   ж) об'єкт індексного дескриптора буде описувати два файли на диску;
   з) об'єкт індексного дескриптора не описуватиме жодного файла на диску.
3. Опишіть, яким чином у реалізації VFS для Linux можна одержати повний шлях до каталогу за номером його індексного дескриптора.
4. Поясніть роботу наступного коду:

```
int main() {
     int fd,n1;
     fd = open( "tmpfile", O_RDWR|O_CREAT|O_TRUNC, 0644 );
     unlink( "tmpfile" );
     n1 = write( fd, "Hello", 5 );
 }
```

   Якщо в цьому коді немає помилки, поясніть, навіщо він може знадобитися.
5. Опишіть послідовність дій, що повинні виконуватися в Linux під час збереження файла на диску в текстовому редакторі. Використана файлова система ext2fs.
6. Як зміниться послідовність дій у попередній вправі, якщо припустити, що замість ext2fs використовують ext3fs?
7. Яким чином файлова система ext2fs забезпечує те, що блоки даних файлів одного каталогу на фізичному рівні розташовують близько один від одного?
8. Чи можна реалізувати підтримку жорстких і символічних зв'язків у файловій системі FAT? Якщо так, то яким чином?
9. Вилучення файла в FAT позначає всі займані ним кластери як вільні, але не очищує їхній вміст. Які при цьому можуть виникнути проблеми?
10. Розробіть застосування для Linux, що відображає тактову частоту процесора.
11. Розробіть застосування для Windows XP, що відображає розмір усіх файлів заданого каталогу до і після стискання. Ім'я каталогу вводить користувач, або його задають у командному рядку.
12. Модифікуйте застосування із попередньої вправи так, щоб ім'я останнього переглянутого каталогу зберігалося в системному реєстрі. Воно має бути використане для перегляду, якщо під час запуску застосування ім'я каталогу не зазначене в командному рядку.

Розділ 14

Виконувані файли

- Статичне компонування виконуваних файлів
- Динамічне компонування та динамічні бібліотеки
- Формат виконуваних файлів ELF
- Розробка динамічних бібліотек у Linux
- Формат виконуваних файлів PE
- Розробка динамічних бібліотек у Windows XP

У цьому розділі розглянемо організацію виконуваного коду в операційних системах. Спочатку зупинимося на створенні виконуваних файлів під час компонування і на тому, як цей процес впливає на структуру таких файлів, потім на понятті динамічної бібліотеки і різних способах завантаження коду таких бібліотек.

При цьому використовуватиметься матеріал кількох попередніх розділів цього підручника.

Код після його завантаження у пам'ять виконують у рамках процесів і потоків, описаних у розділах 3–7. Організація виконуваного коду визначає структуру адресного простору процесу, наведеного в розділі 9. Виконуваний код зберігають у файлах, для читання яких потрібно використовувати інтерфейс файлової системи, а під час їхнього завантаження – технологію відображуваної пам'яті, про що йшлося в розділах 11–13.

14.1. Загальні принципи компонування

Компонуванням (linking) називають процес створення фізичного або логічного виконуваного файла (модуля) із набору об'єктних файлів і файлів бібліотек для подальшого виконання або під час виконання і вирішення проблеми неоднозначності імен, що виникає при цьому.

У разі створення фізичного виконуваного файла для подальшого виконання компонування називають статичним; у такому файлі міститься все потрібне для виконання програми. У разі створення логічного виконуваного файла під час виконання програми компонування називають динамічним; у цьому випадку образ виконуваного модуля збирають «на ходу». Статичне компонування буде темою розділів 14.2–14.3, динамічне – розділу 14.4.

Виокремимо основні питання, які потрібно вирішувати під час реалізації компонування.

- Як давати імена і звертатися в коді до неіснуючих об'єктів на цей момент?
- Як з'єднувати різні простори імен у несуперечливе ціле?

Компонування є реалізацією системи іменування. Такі системи відображають імена на значення, при цьому імена об'єктів у вихідному коді мають відображатися на адреси, необхідні для їхнього використання процесором.

14.2. Статичне компонування виконуваних файлів

14.2.1. Об'єктні файли

Під час компонування виконуваний файл будують із *об'єктних файлів* (object files), які створює компілятор. Об'єктний файл має заголовок, що містить розмір ділянок коду і даних, а також зсув таблиці символів; *об'єктний код* (інструкції і дані, згенеровані компілятором), який звичайно розділений на поіменовані ділянки (секції) залежно від призначення; *таблицю символів* (symbol table).

Приклад створення об'єктних файлів показано на рис. 14.1.

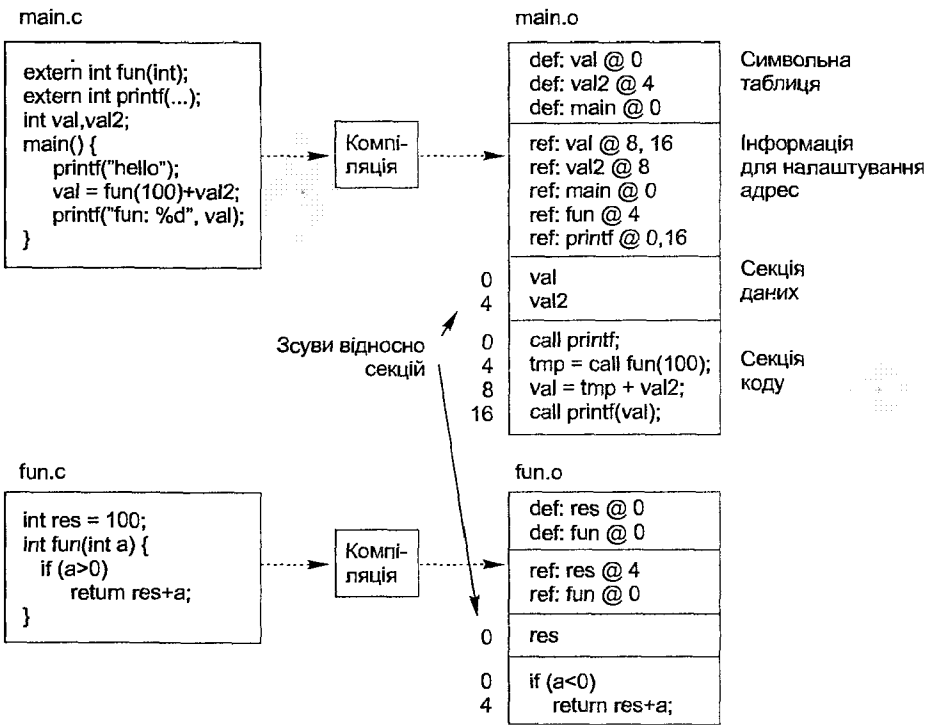

Рис. 14.1. Створення об'єктних файлів

Таблиця символів — це спеціальний розділ об'єктного файла, що містить визначення зовнішніх імен, які задають імена та відносні адреси файлових об'єктів, визначених для використання в інших файлах; зовнішні посилання (глобальні символи, що використовуються у файлі), які зазвичай містять зсув відповідної структції та необхідний символ.

Інформацію про зовнішні посилання називають також інформацією для налаштування адрес (relocation information).

Компілятор створює один об'єктний файл за один запуск, при цьому до інших об'єктних файлів або бібліотек не звертається, тому він ніколи не пов'язує зовнішні посилання із конкретними адресами, тобто не розв'язує їх.

На рис. 14.1, а також на рис. 14.2–14.3 елементи символьної таблиці відображаються так:

`def: name @offset` — інформація про визначення зовнішнього імені `name`, яке перебуває у файлі зі зсувом `offset`;

`ref: name @offset` — інформація про зовнішнє посилання на ім'я `name`, що перебуває у файлі зі зсувом `offset`.

4.2.2. Компонувальники і принципи їх роботи

Як зазначалося, компілятор не розв'язує зовнішні посилання, а отже, не може створити виконуваний файл. Це робота *компонувальника* (linker).

Його основні функції такі: об'єднує всі частини програми у виконуваний файл; збирає разом код і дані секцій одного призначення з різних об'єктних файлів; задає адреси для коду і даних, розв'язуючи при цьому зовнішні посилання.

У результаті за статичного компонування на диск записують виконуваний файл, готовий до запуску, за динамічного — виконуваний файл теж буде створено, але йому для виконання потрібні додаткові файли.

Зазвичай компонувальнику для виконання його роботи потрібні два проходи.

1. На першому він збирає разом секції, розподіляє пам'ять, складає глобальну таблицю символів із елементами, що відповідають кожному використовуваному або визначеному символу. Наприкінці проходу стають відомі адреси всіх секцій виконуваного файла.
2. На другому він коригує кожне посилання в коді з урахуванням інформації про адреси секцій у виконуваному файлі і глобальної таблиці символів, після чого створює виконуваний файл.

Глобальна таблиця символів містить інформацію про програму, що зберігається між проходами компонувальника. За секціями — ім'я, розмір, старе і нове місце розташування; за символами — ім'я, секція, зсув усередині секції.

Зупинимося докладніше на проходах компонувальника.

Головне завдання, яке вирішують на першому проході компонувальника, полягає у визначенні адрес, за якими потрібно розміщувати задані в коді об'єкти. Компілятор під час генерації символьної таблиці не має інформації про те, у якому місці адресного простору процесу потрібно розміщувати дані, а в якому — код; крім того, він вважає, що всі секції починаються від нульової адреси, тому у символьній

таблиці зберігаються пари (`ім'я, зсув`). Їх називають глобальними визначеннями. Завданнями компонувальника під час роботи з ними є:

- визначення розміру і місця розташування кожної секції виконуваного файла і розрахунок адреси розміщення кожного об'єкта всередині цих секцій;
- розміщення всіх об'єктів за їхніми адресами;
- збереження всіх глобальних визначень у глобальній таблиці символів, яка тепер відображатиме визначення об'єктів на їх остаточні віртуальні адреси.

Приклад виконання першого проходу компонувальника показано на рис. 14.2.

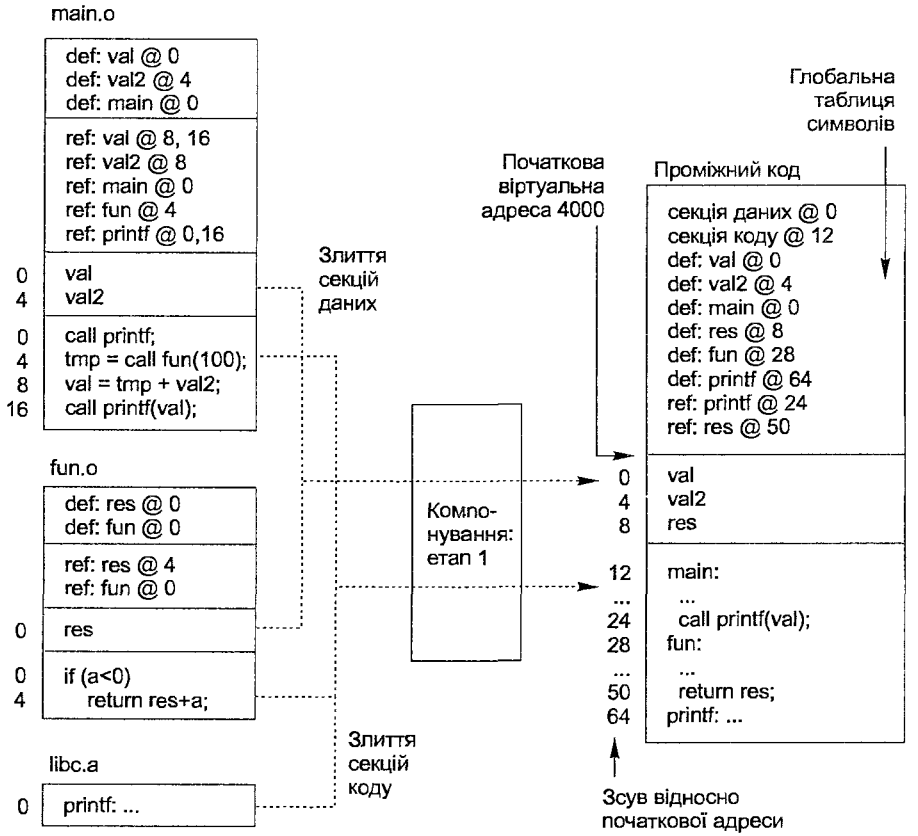

Рис. 14.2. Виконання першого проходу компонувальника

Головне завдання, яке розв'язують на другому проході компонувальника, полягає в корекції всіх адрес в об'єктному коді та розв'язанні всіх зовнішніх посилань. Для цього після збереження всіх посилань у глобальній таблиці символів компонувальник перевіряє, щоб кожний символ мав тільки одне визначення (бути використаний він може скільки завгодно разів). Після цього проходить всіма посиланнями і виконує їхню корекцію, замінюючи посилання адресою відповідного символу.

Процес підстановки адрес замість посилань називають ще налаштуванням адрес (address relocation). Крім основного виду такого налаштування, є інші його варіанти:

- налаштування із базовою адресою і зсувом, коли деякий набір пов'язаних імен, для яких задані зсуви, переводять у набір адрес додаванням зсувів до однієї базової адреси; таке налаштування використовують, наприклад для елементів структур (у сенсі мови С); при цьому звертання до поля структури переводять в адресу додаванням зсуву поля та базової адреси структури;
- корекція статичних даних секцій у разі їхнього переміщення (адреси всіх таких елементів скориговують на величину переміщення).

Приклад виконання другого проходу компонувальника показано на рис. 14.3.

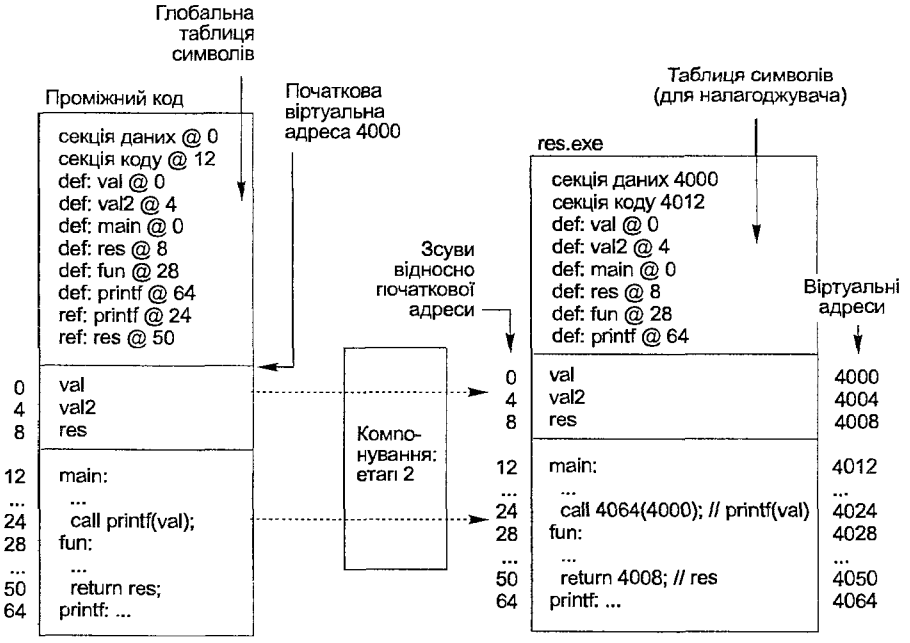

Рис. 14.3. Виконання другого проходу компонувальника

14.3. Завантаження виконуваних файлів за статичного компонування

Виконуваний файл, отриманий внаслідок описаного раніше статичного компонування, містить усе необхідне для створення процесу. Завантаження такого файла у пам'ять виконує окремий компонент ОС – *програмний завантажувач* (program loader). Він звичайно відображає виконуваний файл в адресний простір процесу (здебільшого файл відображають не як єдине ціле, а секціями, причому ділянки пам'яті для секцій виділяє також завантажувач) та ініціалізує керуючий блок процесу таким чином, щоб процес був у стані готовності до виконання.

Найважливішим етапом є відображення виконуваного файла. Використання відображуваної пам'яті для завантаження коду і даних дає змогу реалізувати низку технологій оптимізації такого завантаження.

- Так забезпечують *завантаження на вимогу* (demand loading), коли код не завантажують із диска до його виконання. Під час відображення виконуваного файла в адресний простір процесу спроба виконання коду, не завантаженого у пам'ять, спричинить автоматичне завантаження нової сторінки внаслідок сторінкового переривання.
- Секція даних може виявитися розрідженою і дані не зчитуватимуться із диска, замість них генеруватимуться сторінки, заповнені нулями.
- Так полегшують реалізацію спільного використання сторінок коду кількома процесами.
- Якщо кеш файлової системи використовує відображувану пам'ять (як описано у розділі 12.3.2), сторінки виконуваного файла після завантаження потраплятимуть у цей кеш.

Зазначимо, що під час відображення виконуваного файла у пам'яті автоматично розміщають його код та ініціалізовані дані. Стек і динамічну ділянку пам'яті зазвичай створюють заново, при цьому для динамічної ділянки компілятор і компонувальник можуть тільки задати її початок, а всю інформацію із керування стеком визначає компілятор із використанням адресації щодо покажчика стека (який буде встановлено завантажувачем).

14.4. Динамічне компонування

14.4.1. Поняття динамічної бібліотеки

Статичне компонування виконуваних файлів має низку недоліків.

- Якщо кілька застосувань використовують спільний код (наприклад, код бібліотеки мови С), кожний виконуваний файл міститиме окрему копію цього коду; у результаті такі файли займатимуть значне місце на диску і у пам'яті. Хотілося б мати можливість зберігати на диску і завантажувати у пам'ять тільки одну копію спільного коду.
- Під час кожного оновлення застосування потрібно перекомпілювати, перекомпонувати і перевстановити.
- Неможливо реалізувати динамічне завантаження програмного коду під час виконання (наприклад, якщо потрібно реалізувати модульну структуру програми, подібно до того, як це зроблено у ядрі Linux).

Для вирішення цих і подібних проблем було запропоновано концепцію динамічного компонування із використанням динамічних або розділюваних бібліотек (dynamic-link libraries (DLL), shared libraries).

Динамічна бібліотека – набір функцій, скомпонованих разом у вигляді бінарного файла, який може бути динамічно завантажений в адресний простір процесу, що використовує ці функції. *Динамічне завантаження* (dynamic loading) – завантаження під час виконання процесу (зазвичай реалізоване як відображення

файла бібліотеки в його адресний простір), *динамічне компонування* (dynamic linking) – компонування образу виконуваного файла під час виконання процесу із використанням динамічних бібліотек.

14.4.2. Переваги і недоліки використання динамічних бібліотек

Переваги використання динамічних бібліотек (для стислості вживатимемо також термін «DLL», який частіше застосовують для позначення динамічних бібліотек у Windows-системах) наведено нижче.

- Оскільки бібліотечні функції містяться в окремому файлі, розмір виконуваного файла стає меншим. Якщо врахувати, що є динамічні бібліотеки, які використовують майже всі застосування у системі (стандартна бібліотека мови C у Linux, бібліотека підсистеми Win32 у Windows XP), то очевидно, що так заощаджують дуже багато дискового простору.
- Якщо динамічну бібліотеку використовують кілька процесів, у пам'ять завантажують лише одну її копію, після чого сторінки коду бібліотеки відображаються в адресний простір кожного з цих процесів. Це дає змогу ефективніше використовувати пам'ять.
- Оновлення застосування може бути зведене до встановлення нової версії динамічної бібліотеки без необхідності перекомпонування тих його частин, які не змінилися.
- Динамічні бібліотеки дають змогу застосуванню реалізувати динамічне завантаження модулів на вимогу. На базі цього може бути реалізований розширюваний API застосування. Для додавання нових функцій до такого API стороннім розробникам достатньо буде створити і встановити нову динамічну бібліотеку, яка підлягає певним правилам.
- Динамічні бібліотеки дають можливість спільно використовувати ресурси застосування (наприклад, така бібліотека може містити спільний набір піктограм), крім того, вони дають змогу спростити локалізацію застосування (якщо всі рядки, які використовуються програмою, помістити в окрему DLL, для заміни мови застосування достатньо буде замінити тільки цю DLL).
- Оскільки динамічні бібліотеки є двійковими файлами, можна організувати спільну роботу бібліотек, розроблених із використанням різних мов програмування і програмних засобів, що спрощує створення застосувань на основі програмних компонентів (отже, динамічне компонування лежить в основі компонентного підходу до розробки програмного забезпечення).

Динамічні бібліотеки не позбавлені недоліків (хоча вони тільки в окремих випадках виправдовують використання статичного компонування).

- Використання DLL сповільнює завантаження застосування. Що більше таких бібліотек потрібно процесу, то більше файлів треба йому відобразити у свій адресний простір під час завантаження, а відображення кожного файла забирає час. Для прискорення завантаження рекомендують укрупнювати DLL, об'єднуючи кілька взаємозалежних бібліотек в одну загальну.

- У деяких ситуаціях (наприклад, під час аварійного завантаження системи із дискети) використання спільних системних DLL неприйнятне через нестачу дискового простору для їхнього зберігання (такі системні DLL, подібно до стандартної бібліотеки мови С, можуть займати кілька мегабайтів дискового простору, при цьому застосування часто потребують усього по кілька функцій із них). У такій ситуації найчастіше використовують версії застосувань, статично скомпоновані таким чином, щоб у їхні виконувані файли був включений зі стандартних бібліотек код лише тих функцій, які їм потрібні.
- Найбільшою проблемою у використанні динамічного компонування є проблема зворотної сумісності динамічних бібліотек. Розглянемо її окремо.

Зворотна сумісність динамічних бібліотек

Ця проблема виникає в ситуації, коли застосування встановлює нову версію DLL поверх попередньої. Якщо нова версія не має зворотної сумісності із попередніми, застосування, розраховані на використання попередніх версій бібліотеки, можуть припинити роботу. Досягти такої сумісності досить складно, особливо коли попередня версія містила відомі помилки, і застосування, що використовують бібліотеку, розробили код їхнього обходу – виправлення помилки у бібліотеці може зробити код застосування невірним (кажуть, що в цьому разі порушується сумісність за помилками – bug-to-bug compatibility).

Деякі ОС (переважно це стосується Windows-систем, але певні проблеми є й в UNIX) ускладнювали цю проблему через те, що не давали можливості кільком версіям однієї й тієї самої бібліотеки одночасно бути завантаженими у пам'ять; виділяли для динамічних бібліотек усіх застосувань спільний каталог, цим «запрошуючи» застосування перезаписувати динамічні бібліотеки один одного (можлива була навіть ситуація, коли стару версію бібліотеки записували поверх нової); не зберігали в динамічних бібліотеках і застосуваннях інформації про точні версії бібліотек, від яких вони залежать.

Усе це призвело до ситуації, котра стосовно Windows-систем (насамперед Consumer Windows) дістала назву «пекло DLL» (DLL hell), коли із часом виявлялося неможливо визначити, що за версія бібліотеки була встановлена і яким застосуванням і що за версія і якому застосуванню потрібна насправді. По суті не було способу гарантовано забезпечити використання застосуванням тієї версії бібліотеки, з розрахунком на яку воно розроблялося.

Розроблювачі сучасних ОС намагаються виправити цю ситуацію (блокуванням і резервним копіюванням важливих DLL, дозволом кільком версіям DLL бути одночасно завантаженими у пам'ять, а також підтримкою використання застосуванням тільки бібліотек із його робочого каталогу).

14.4.3. Неявне і явне зв'язування

Є два основні способи завантаження динамічних бібліотек в адресний простір процесу – *неявне* і *явне зв'язування* (implicit і explicit binding).

Неявне – основний спосіб завантаження динамічних бібліотек у сучасних ОС. При цьому бібліотеку завантажують автоматично до початку виконання застосування під час завантаження виконуваного файла, за це відповідає завантажувач

виконуваних файлів ОС. У деяких системах такий завантажувач є частиною ядра ОС, у деяких — окремим застосуванням. Список бібліотек, потрібних для завантаження, зберігають у виконуваному файлі.

До переваг цього методу належать:

- простота і прозорість з погляду програміста (йому не потрібно писати код завантаження бібліотек, достатньо у налаштуваннях компонувальника вказати список бібліотек, які йому потрібні);
- висока ефективність роботи процесу після початкового завантаження (усі необхідні бібліотеки до цього часу вже завантажені у його адресний простір).

Недоліком неявного зв'язування можна вважати зниження гнучкості (так, наприклад, якщо хоча б однієї з необхідних бібліотек не буде на місці, процес завантаження не обійдеться без проблем, навіть коли для виконання конкретної задачі ця бібліотека не потрібна). Крім того, збільшуються час завантаження і початковий обсяг необхідної пам'яті.

Альтернативним для неявного є явне зв'язування, коли динамічну бібліотеку завантажують в адресний простір процесу виконанням системного виклику із його коду. Після цього, використовуючи інший системний виклик, застосування отримує адресу необхідної йому функції бібліотеки і може її викликати. Після використання бібліотеку можна вилучити з пам'яті. Компонувальник при цьому нічого про неї не знає, завантажувач ОС автоматично бібліотек не завантажує (здебільшого неявне зв'язування зводиться до автоматичного виконання тих самих викликів, які сам програміст виконує за явного).

Такий підхід вимагає від програміста додаткових зусиль, але має більшу гнучкість. Однак складність реалізації призводить до того, що його використовують лише тоді, коли застосуванню справді потрібно завантажувати і вивантажувати додаткові бібліотеки під час виконання.

14.4.4. Динамічні бібліотеки та адресний простір процесу

Після того, як динамічна бібліотека була відображена в адресний простір процесу, вона стає майже прозорою для програмного коду, що тут виконується.

Усі функції бібліотеки стають доступними для всіх потоків цього процесу, фактично її код і дані набувають вигляду доданих до адресного простору процесу. Зазначимо, що під час відображення бібліотеки у пам'ять використовують технологію копіювання під час записування, тому кожен процес матиме свою копію стека і даних бібліотеки.

З іншого боку, для коду бібліотечної функції будуть доступні такі ресурси, як дескриптори відкритих файлів процесу і стек потоку, що викликав дану функцію. Не слід, однак, забувати про те, що під час роботи із даними потоку із коду бібліотеки потрібно виявляти обережність, зокрема, ніколи не вивільняти пам'ять, розподілену не в цій бібліотеці. Іншими словами, код бібліотек, розрахованих на використання у багатопотокових застосуваннях, має бути безпечним з погляду потоків (thread-safe).

14.4.5. Особливості об'єктного коду динамічних бібліотек

Код динамічних бібліотек звичайно зберігають у виконуваних файлах формату, стандартного для цієї ОС (далі побачимо, що виконувані файли і файли DLL можуть відрізнятися тільки одним бітом у заголовку), але з погляду характеру цього коду є одна важлива відмінність між кодом DLL і кодом звичайних виконуваних файлів. Вона полягає в тому, що код DLL в один і той самий час повинен мати можливість завантажуватися за різними адресами. Для того щоб це було можливо, такий код потрібно робити позиційно-незалежним.

Позиційно-незалежний код завжди використовує відносну адресацію (базову адресу додають до зсуву). Базову адресу налаштовують у момент завантаження DLL в адресний простір процесу і називають також базовою адресою бібліотеки; такі адреси відрізняються для різних процесів. Зсув у цьому разі називають внутрішнім зсувом об'єкта.

14.4.6. Точка входу динамічної бібліотеки

Одна із функцій динамічної бібліотеки може бути позначена як її *точка входу*. Така функція автоматично виконуватиметься завжди, коли цю DLL відображають в адресний простір процесу (явно або неявно); у неї можна поміщати код ініціалізації структур даних бібліотеки. Багато систем дають змогу задавати також і функцію, що викличеться в разі вивантаження DLL із пам'яті.

14.5. Структура виконуваних файлів

У сучасних ОС є тенденція до спрощення процедури завантаження виконуваних файлів через наближення їхнього формату до образу процесу у пам'яті. Описати загальну структуру виконуваного файлу доволі складно, у цьому розділі зупинимося на деяких спільних компонентах таких файлів, а потім розглянемо конкретні формати.

Оскільки виконувані файли створює компонувальник на базі об'єктних файлів, то у структурі цих файлів є багато спільного. Насамперед це стосується того, що обидва види файлів складаються з набору секцій різного призначення.

Деякі спільні елементи структури виконуваних файлів (їхній порядок і точний зміст розрізняють для різних форматів цих файлів) наведено нижче.

- Насамперед виконувані файли мають заголовок. Він найчастіше містить «магічні символи», які дають змогу ОС швидко визначити, що цей файл є виконуваним конкретного типу; базову адресу відображення виконуваного коду у пам'ять; ознаку відмінності між незалежним виконуваним файлом і DLL; адреси найважливіших елементів файла.
- Майже завжди в такі файли включають інформацію для динамічного компонувальника. Звичайно ця інформація складається зі *списку імпорту*, що містить інформацію про всі DLL, потрібні для виконання цього файла (для неявного зв'язування) та *списку експорту*, що містить інформацію про всі функції, доступні для використання іншими виконуваними файлами.

- Деякі формати виконуваних файлів використовують зовнішній динамічний завантажувач; у цьому разі всередині файла також зберігають інформацію про його місцезнаходження.
- Інформацію про всі секції файла зберігають у списку секцій, який називають таблицею секцій (section table). Елементи цієї таблиці описують різні секції файла.
- Нарешті, у файлі розташовані самі секції, що містять дані різного призначення. Назви секцій та їхній вміст розрізняються для різних форматів, але майже завжди є секції для коду та ініціалізованих даних.

14.6. Виконувані файли в Linux

14.6.1. Формат ELF

Формат ELF [71] є основним форматом виконуваних файлів для Linux та інших сучасних UNIX-систем. Цей формат можна використати для таких типів файлів:

- об'єктних, призначених для статичного компонування (їх також називають переміщуваними файлами);
- виконуваних, котрі описують, яким чином виклик `exec()` створює образ процесу;
- розділюваних (динамічних) бібліотек, які динамічний компонувальник збирає в образ процесу.

Загальну структуру файла у форматі ELF показано на рис. 14.4.

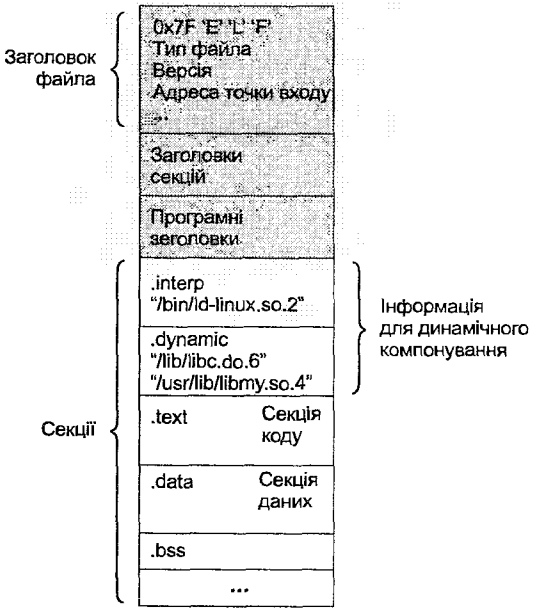

Рис. 14.4. Структура файла в ELF-форматі

Заголовок файла

Заголовок файла у форматі ELF описує його організацію. Він починається із чотирьох обов'язкових «магічних» символів 0x7F, 'E', 'L', 'F'; крім цього, містить інформацію про тип файла (об'єктний файл, виконуваний файл, динамічна бібліотека), архітектуру процесора, версію файла, адресу точки входу (за яким ОС передасть керування після завантаження), нарешті, про зсув у файлі таблиці програмних заголовків і таблиці заголовків секцій (ці таблиці розглянемо окремо).

Секції файла

Інформація про всі секції ELF-файла перебуває у *таблиці заголовків секцій*. Вона містить заголовки секцій, кожен із яких описує одну секцію і включає її ім'я, тип, розмір, адресу, на яку вона має відображатися, інформацію про те, чи дозволене записування у секцію після її завантаження у пам'ять.

Розглянемо деякі наперед визначені секції ELF-файла: `.text` містить виконуваний код програми, `.data` і `.data1` — ініціалізовані дані; `.bss` призначена для організації сегмента неініціалізованих даних у пам'яті; вона реалізована як «діра» у розрідженому файлі, після завантаження якого у пам'ять спроба звернутися до даних секції спричиняє повернення сторінки, заповненої нулями; `.symtab` містить таблицю символів цього файла; `.strtab` — таблицю рядків цього файла, в якій розташовані використовувані в коді програми рядкові константи.

Об'єктні файли містять також спеціальну секцію `.reloc` із даними для налаштування адрес. Крім цього, можна виокремити кілька секцій, що зберігають інформацію для системного завантажувача і динамічного компонувальника, що буде розглянуто окремо. Застосування можуть задавати і свої власні секції.

Зазначимо, що в термінології ELF секція означає поіменований розділ у виконуваному файлі; після відображення у пам'ять секції відповідає сегмент.

Програмний заголовок

ELF-файли містять інформацію, яка буде використана для створення образу процесу після завантаження цього файла у пам'ять. Кажуть про два відображення інформації у форматі ELF — як дискового файла і як образу процесу. Структурою даних, що керує завантаженням ELF-файлів, є програмний заголовок (program header).

Після завантаження у пам'ять на базі секцій ELF-файлу створюють сегменти пам'яті, причому один сегмент може відповідати кільком секціям. Програмний заголовок — це масив елементів, кожен із яких описує один сегмент. Такий елемент може містити тип сегмента (код, дані тощо), його зсув від початку файла, віртуальну адресу початку сегмента в пам'яті, розмір сегмента на диску і в пам'яті. Інформацію про динамічне компонування завантажують в особливі сегменти.

Базову адресу бібліотеки або виконуваного файла обчислюють на основі адреси завантаження та віртуальної адреси першого завантаженого сегмента, визначеного в програмному заголовку (при цьому його округляють до розміру сторінки).

14.6.2. Динамічне компонування в Linux

Спочатку розглянемо елементи ELF-файла, відповідальні за динамічне компонування, а потім перейдемо до опису всього процесу такого компонування бібліотек формату ELF.

Структури даних підтримки динамічного компонування

Для того щоб забезпечити динамічне компонування, ELF-файл містить ряд структур даних. Розглянемо деякі з них.

- Шлях до динамічного компонувальника задають тому, що ELF-формат передбачає реалізацію такого компонувальника окремою програмою. Цей шлях розміщають у спеціальній секції `.interp` і завантажують в особливий сегмент під час виконання. Про принципи роботи такого компонувальника йтиметься окремо.
- Інформацію про необхідні динамічні бібліотеки зберігають у спеціальній секції `.dynamic`, там також міститься інформація про точку входу динамічної бібліотеки.

Зазначимо, що в ELF-файлі для динамічних бібліотек не задають окремої інформації про набір функцій, експортованих цією бібліотекою, вважаючи, що бібліотека експортує всі функції, інформація про які наявна у її таблиці символів.

Імена і розташування динамічних бібліотек

До розділюваних бібліотек можна звертатися в різних ситуаціях за трьома різними іменами.

- *Основним* (soname). Використовується динамічним компонувальником і має таку структуру: libname.so.N, де name — ім'я бібліотеки, N — базовий номер версії (зміна цього номера зазвичай пов'язана із несумісними змінами в інтерфейсі бібліотеки). Повне основне ім'я включає каталог, у якому перебуває бібліотека. Приклад основного імені: libdl.so.1.
- *Реальним* (real name). Це ім'я файла, у якому зберігається виконуваний код бібліотеки. Воно додає до основного імені .M1.M2, де M1 — молодший номер версії, M2 — номер реалізації (останній можна пропускати). Прикладом реального імені може бути libdl.so.1.9.5. Повне основне ім'я задають як символічний зв'язок, що вказує на цей файл.
- *Для компонувальника* (linker name). Використовують під час компонування застосування, що потребує бібліотеки; воно — основне ім'я без номера версії: libdl.so. Таке ім'я задають як символічний зв'язок, що вказує на повне основне ім'я. Компонувальник отримує ім'я libxx.so у вигляді параметра -lxx, для libdl.so цей параметр матиме такий вигляд: -ldl.

Ось фрагмент відображення вмісту каталогу, на якому видно всі три імені бібліотеки:

```
lrwxrwxrwx ... libdl.so -> libdl.so.1*
lrwxrwxrwx ... libdl.so.1 -> libdl.so.1.9.5*
-rwxr-xr-x ... libdl.so.1.9.5*
```

Під час розробки бібліотеки створюють файл із реальним іменем. Коли встановлюють нову версію динамічної бібліотеки, його поміщають в один із наперед визначених каталогів, після чого запускають спеціальну утиліту `ldconfig`, яка пе-

ревіряє наявні файли й автоматично створює символічні зв'язки для основних імен на підставі інформації про версії з ELF-заголовка. Ця утиліта також обновлює кеш динамічного компонувальника, який розглянемо пізніше.

Зв'язки, що відповідають іменам для компонувальника, задають вручну.

Використання динамічних бібліотек із застосувань

Коли компонують застосування, що використовує динамічну бібліотеку, необхідно вказати ім'я для компонувальника, яке посилається на основне ім'я. Основне ім'я збережеться у виконуваному файлі застосування у списку необхідних бібліотек (у секції `.dynamic`). Жодних додаткових дій у коді застосування виконувати не потрібно — усі функції бібліотеки будуть доступні в ньому як глобальні функції.

Ось приклад запуску компілятора `gcc` для збирання застосування, яке використовує бібліотеку з іменем для компонувальника **libabc.so**, що перебуває в каталозі **/usr/local/lib**:

```
$ gcc myprog.c -o myprog -labc -L /usr/local/lib
```

Розробка динамічних бібліотек

Динамічні бібліотеки в Linux створюють за допомогою стандартного компілятора мови C, подібно до будь-якого виконуваного файла. Тут докладно не розглядатимемо цей процес (він описаний, наприклад, в [24] і [107]), зазначимо тільки, що компонувальнику треба вказати на необхідність генерації позиційно-незалежного коду (для `gcc` це параметр `-fpic`), а також задати ім'я, яке потрібно використати як основне ім'я бібліотеки (soname). Для `gcc` його треба задавати так: `-Wl,-soname,основне_ім'я`.

Наведемо приклад запуску компілятора `gcc` для збирання файла бібліотеки **libabc.so.1.0.0** з основним іменем **libabc.so.1**:

```
$ gcc -shared -o libabc.so.1.0.0 -fpic -Wl,-soname,libabc.so.1 abc.c
```

Розробка коду бібліотеки не потребує додаткових дій — усі створені глобальні функції будуть доступні для застосувань, що використовують бібліотеку.

Точка входу у бібліотеку має реалізовуватись як функція `_init()`. Її викликатимуть щоразу під час відображення бібліотеки у пам'ять процесу. Функція, яку викликатимуть у разі вивантаження бібліотеки із пам'яті, називається `_fini()`.

Динамічний компонувальник і неявне зв'язування бібліотек

Запуск виконуваного файла у форматі ELF призводить до того, що керування отримує *динамічний компонувальник*, шлях до якого зазначений у секції `.interp` ELF-файла. У Linux такий компонувальник називають **/lib/ld-linux.so.N** (*N* — номер версії). Він у свою чергу відшукує і завантажує всі динамічні бібліотеки, потрібні для виконання застосування, використовуючи список необхідних бібліотек (секцію `.dynamic`); під час пошуку він використовує основні імена. Так реалізоване неявне зв'язування.

Список каталогів, у яких динамічний компонувальник має шукати бібліотеки, задають у його конфігураційному файлі **/etc/ld.so.conf**. Стандартними каталогами для динамічних бібліотек є **/lib** і **/usr/lib**, пошук у них здійснюють, навіть якщо не заданий конфігураційний файл.

Пошук у всіх цих каталогах під час кожного запуску застосування неефективний, тому організовують кеш імен динамічних бібліотек і зберігають у файлі /etc/ld.so.cache. Його створює утиліта ldconfig після того, як встановить усі символічні зв'язки. У разі необхідності шлях до потрібної бібліотеки вибирають із кеша, що підвищує ефективність завантаження застосувань. На жаль, підтримка коректного стану кеша вимагає ручного запуску ldconfig під час операцій додавання, вилучення або зміни будь-якої динамічної бібліотеки, інакше динамічний компонувальник не врахує цих змін. Це видається недостатньо зручним.

Можна змусити застосування використати для конкретного запуску іншу версію динамічної бібліотеки. Для цього використовують змінну оточення LD_LIBRARY_PATH, що є списком каталогів. За наявності такої змінної саме із каталогів цього списку починає пошук бібліотек динамічний компонувальник. Якщо в одному із каталогів цього списку буде розміщена інша версія бібліотеки, саме вона буде використана замість стандартної версії, наприклад з /usr/lib.

Ця змінна зручна також тоді, коли зміна у бібліотеці, що не зачіпає основного імені, призвела до порушення роботи деякого застосування. Для цього потрібно перенести стару версію бібліотеки в інший каталог і задати перед запуском застосування значення LD_LIBRARY_PATH, що включає каталог зі старою версією.

Дізнатися, які динамічні бібліотеки потрібні застосуванню, можна за допомогою утиліти ldd, параметром якої є ім'я виконуваного файла.

Явне зв'язування динамічних бібліотек

Для явного завантаження бібліотеки під час виконання застосування та виклику її функцій потрібно виконати кілька кроків.

1. Завантажити бібліотеку за допомогою системного виклику dlopen(libpath, flags). Як параметри цей виклик приймає:
   - шлях до бібліотеки libpath (у разі задання повного шляху виклик відразу знаходить файл, якщо задано тільки ім'я — здійснює пошук, аналогічний до того, що виконує динамічний компонувальник);
   - набір прапорців flags, які керують розв'язанням посилань під час завантаження (наприклад, вмикання прапорця RTLD_NOW означає, що всі зовнішні посилання мають розв'язуватися під час завантаження і в разі відмови у розв'язанні хоча б одного посилання завантаження не відбудеться).

   Цей виклик повертає дескриптор бібліотеки, який використовується в інших функціях.
2. Знайти символ у бібліотеці за іменем шляхом виклику dlsym(libd, sym). Першим параметром цього виклику є дескриптор бібліотеки, другим — ім'я символу. Цей виклик повертає адресу символу (функції) у бібліотеці, через цей покажчик можна викликати бібліотечну функцію.
3. Закрити дескриптор бібліотеки за допомогою виклику dlclose(libd). Бібліотеку вивантажать із пам'яті, якщо для неї не залишиться жодного відкритого дескриптора.

Ось приклад завантаження бібліотеки і виклику функції з неї:

```
#include <dlfcn.h>
typedef int(*fint)(int);
```

```
fint fun;      // оголошення покажчика на функцію
void *libd = dlopen("libmy.so.1", RTLD_NOW);
if (libd) {
    fun = dlsym(libd, "fun");
    int res = (*fun)(100);  // виклик функції через покажчик
    dlclose(libd);
}
```

14.6.3. Автоматичний виклик інтерпретаторів

Під скриптами звичайно розуміють програми, написані на різних інтерпретованих мовах – Perl, Python, TCL, мові командного інтерпретатора UNIX (shell-скрипти). Звичайний спосіб запуску таких скриптів потребує явного задання імені інтерпретатора в командному рядку

```
$ perl test.pl
```

У цьому разі буде викликано інтерпретатор мови Perl, а йому на вхід подано Perl-скрипт `test.pl`, після чого інтерпретатор починає виконувати цей скрипт. Зазначати інтерпретатор щоразу під час запуску скрипта не зовсім зручно.

У всіх UNIX-системах є спосіб виконувати такі файли без вказання імені інтерпретатора в командному рядку. Ця можливість реалізована на рівні завантажувача виконуваних файлів.

Для того щоб файл скрипта був поданий завантажувачем виконуваних файлів на вхід інтерпретатору автоматично, у першому рядку цього файла треба зазначити повний шлях до інтерпретатора після комбінації символів #! і, крім того, для цього файла задати права на виконання.

Ось приклад скрипта на Perl, яка використовує цю технологію:

```
#!/usr/bin/perl
# далі йде звичайний текст скрипта test.pl
```

Спробу виконання такого файла перехоплює завантажувач, який завантажує у пам'ять заданий інтерпретатор, а як параметр командного рядка передає йому файл, запуск якого він перехопив.

14.7. Виконувані файли у Windows XP

14.7.1. Формат PE

Файли формату PE (Portable Executable) [89] з'явилися разом із Windows NT. Вони мають багато спільного із традиційними форматами UNIX-систем (об'єктні файли, на основі яких будують такий файл, можуть бути у форматі COFF – одному зі стандартних форматів об'єктних файлів у UNIX). У цьому розділі торкнемося загальних рис PE-формату і формату ELF, а також докладніше розглянемо найістотніші відмінності між ними. Загальна структура файла в форматі PE наведена на рис. 14.5.

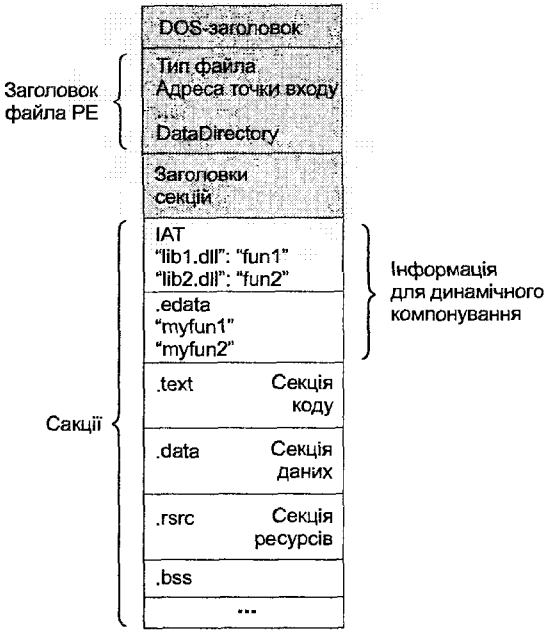

Рис. 14.5. Структура файла у форматі PE

Основна подібність між двома форматами полягає у їхній секційній організації та описі списку секцій таблицею секцій. Зазначимо, що навіть назви і призначення багатьох основних секцій збігаються — це стосується `text`, `.data`, `.bss`, `.reloc`. Окрема секція, як і в ELF, виділена під опис списку імпорту.

Основні відмінності між двома форматами наведені нижче.

- Перед основним заголовком PE-файла поміщають так званий DOS-заголовок. Він містить невелику програму, яка у разі запуску під MS-DOS виводить повідомлення і завершується. Цей заголовок залишився відтоді, коли PE-файли часто намагалися запускати під керуванням MS-DOS або Windows 3.x. Під час звичайного запуску (під Windows XP) керування негайно передають за адресою початку виконання, заданою в основному заголовку.
- Інформацію, необхідну для виконання файла, не виносять в окрему структуру даних, а зберігають у стандартних структурах — основному заголовку, заголовках секцій тощо.
- Для зручності доступу адреси найважливіших об'єктів усередині файла (секцій експорту, імпорту, ресурсів тощо) утримують у заголовку файла як окремий масив DataDirectory.
- Окрема секція `.rsrc` зарезервована для зберігання ресурсів програми. Ресурси всередині цієї секції мають деревоподібну організацію із каталогами і підкаталогами.

Найсуттєвішими є відмінності у структурі інформації для динамічного компонування, про які йтиметься в наступному розділі.

14.7.2. Динамічне компонування у Windows XP

Формат PE і динамічне компонування

Спочатку розглянемо відмінності підтримки динамічного компонування у форматі PE від підтримки формату ELF.

- У Windows XP динамічне компонування здійснює система, а не окремий динамічний компонувальник; очевидно, що всередині виконуваного файла ім'я такого компонувальника не задають.
- Окрема секція .edata виділена для опису експортованих символів. Це досить важлива відмінність від ELF, для якого всі символи за замовчуванням є експортованими.
- Відмінності має і секція імпортованих функцій. Тут створена спеціальна таблиця, що визначає всі імпортовані функції; після запуску її заповнюють адресами функцій з DLL. Таку таблицю називають таблицею імпорту адрес (IAT). Використання IAT дає змогу звести всю інформацію про імпортовані адреси в одне місце у файлі.

Зазначимо, що як і для ELF, відмінностей між динамічними бібліотеками і виконуваними файлами із погляду формату файла немає, фактично їх розрізняють за значенням поля типу в заголовку.

Процес компонування у разі неявного зв'язування

Для реалізації неявного зв'язування все, що потрібно від розробника застосування, — це вказати компонувальнику список необхідних DLL і впевнитися, що під час виконання всі вони можуть бути знайдені. Завантажувач ОС забезпечить пошук і виконання потрібного коду. Зауважимо, що якщо під час завантаження DLL виникла помилка, весь процес завантаження переривається. До початку виконання основного потоку процесу всі необхідні DLL мають бути відображені в його адресний простір. У розробці DLL [31], на відміну від UNIX, основним завданням програміста є задання списку експортованих функцій. Цього можна домогтися такими способами:

- створити спеціальний файл із розширенням .DEF, у якому перелічити всі такі функції, і передати його компонувальнику;
- у разі використання Visual C++ скористатися спеціальною конструкцією _declspec(dllexport), яку потрібно поміщати перед оголошенням експортованої функції:

```
_declspec(dllexport) DWORD fun() {
    printf("Виклик fun()\n");
    return 100;
}
```

При цьому додатково до створення DLL компонувальник згенерує спеціальний файл заглушок — статичну бібліотеку із розширенням .LIB, яку компонують із клієнтським застосуванням і яка містить код заглушок для створення зв'язків із DLL під час завантаження. Ім'я цієї бібліотеки має бути явно задане як один із параметрів виклику компонувальника під час компонування клієнтського застосування.

Компонуючи DLL за допомогою Visual C++, потрібно вмикати прапорець -LD:

```
c:\mydll> cl -LD -o mydll.dll mydll.c
```

У разі використання функцій із DLL їх, на відміну від UNIX, потрібно імпортувати. Це може мати такий вигляд:

```
_declspec(dllimport) DWORD fun();
void main() {
    printf("%d\n", fun());
}
```

Тоді під час компонування будуть узяті функції із .LIB-файла, а в разі виконання заглушки звертатимуться до справжніх функцій із DLL.

Виклик компілятора Visual C++ для компонування клієнтського застосування, що використовує DLL, матиме вигляд:

```
c:\dllclient> cl -o dllclient dllclient.c c:\mydll\mydll.lib
```

Точка входу в DLL

Точку входу у DLL у Win32 API описують як функцію:

```
BOOL WINAPI DllMain(HINSTANCE libh, DWORD reason, LPVOID reserved);
```

Першим параметром для неї є дескриптор екземпляра бібліотеки, другим — індикатор причини виклику (DLL_PROCESS_ATTACH — під час завантаження бібліотеки, DLL_PROCESS_DETACH — у разі її вивантаження), третій параметр не використовують.

```
BOOL WINAPI DllMain(HINSTANCE libh, DWORD reason, LPVOID reserved) {
    switch (reason) {
        case DLL_PROCESS_ATTACH: printf("завантаження DLL\n"); break;
        case DLL_PROCESS_DETACH: printf("вивантаження DLL\n"); break;
    }
    return TRUE;
}
```

Відкладене завантаження DLL

Для прискорення завантаження застосувань Windows XP надає можливість відкладеного завантаження DLL. У цьому разі бібліотеку пов'язують із виконуваним файлом неявно, але завантажують у пам'ять тільки під час першого звертання до одного із символів, визначених у ній. Для того щоб задати відкладене завантаження для DLL під час компонування клієнта, потрібно вказати її ім'я як аргумент параметра компонувальника DelayLoad; крім того, необхідно скомпонувати із проектом бібліотеку delayimp.lib.

```
C:\dllclient>cl -o dllclient dllclient.c c:\mydll\mydll.lib
     delayimp.lib -link -DelayLoad:mydll.dll
```

Процес розробки самої бібліотеки залишається незмінним.

Явне зв'язування

Процес явного зв'язування DLL у Windows XP складається переважно з тих самих кроків, що і для Linux.

1. Для того щоб відобразити DLL в адресний простір процесу, використовують функцію LoadLibrary(libpath) з одним параметром, який задає шлях до файла

бібліотеки. Ця функція повертає дескриптор екземпляра бібліотеки (значення типу HINSTANCE). Вона є аналогом dlopen().

2. Аналогом dlsym() для отримання покажчика на функцію за її іменем є функція GetProcAddress(libh, sym). Її параметри за змістом ті самі, що і для dlsym() — дескриптор екземпляра і рядок з іменем функції.
3. Для вивантаження бібліотеки із пам'яті використовують функцію FreeLibrary(libh).

Ось приклад застосування явного зв'язування у Win32 API (він майже нічим не відрізняється від прикладу для Linux):

```
typedef int(*fint)(int);
fint fun;
HINSTANCE libh = LoadLibrary("my.dll");
if (libh) {
     fun = (fint) GetProcAddress(libh,"fun");
     int res = fun(100);
     FreeLibrary(libh);
}
```

14.7.3. Зворотна сумісність DLL у Windows 2000 і Windows XP

У цьому розділі розглянемо, як вирішують проблему зворотної сумісності динамічних бібліотек в останніх версіях ОС лінії Windows XP.

Переспрямування DLL

Для того щоб вирішити проблему засмічення системних каталогів динамічними бібліотеками застосувань, у Windows 2000 з'явилася можливість змусити завантажувач спочатку переглядати під час пошуку необхідних DLL робочий каталог застосування. Таку можливість називають *переспрямуванням DLL* (DLL redirection), для її реалізації достатньо помістити в робочий каталог застосування файл із іменем, отриманим із імені виконуваного файла додаванням суфікса .local (наприклад, MyApp.exe.local, якщо виконуваний файл називають MyApp.exe). Вміст файла ролі не відіграє.

Ізольовані застосування і паралельні збірки

У Windows XP запропоновано повніше вирішення даної проблеми. У цій системі з'явилася можливість розробляти застосування, залежність яких від зовнішніх компонентів задають явно, — ізольовані застосування (isolated applications).

Опис залежностей ізольованого застосування зберігають у спеціальному файлі маніфесту застосування (MyApp.exe.manifest). У ньому перераховані компоненти, від яких залежить це застосування. Кожен такий компонент відображають паралельною збіркою (side-by-side assembly) — набором взаємозалежних ресурсів, описаних файлом маніфесту збірки (assembly manifest). Звичайно збірку відображають окремою DLL.

Кожна збірка має версію, при цьому у Windows XP можливе одночасне завантаження у пам'ять кількох версій однієї й тієї самої збірки. За це відповідає спеціальний компонент динамічного завантажувача ОС — менеджер паралельного

завантаження (side-by-side manager). Для визначення правильності зв'язування використовують інформацію із файла маніфесту застосування. Якщо в маніфесті описана залежність від конкретної версії збірки, завантажують цю версію, інакше – версію за замовчуванням.

Якщо задано конкретну версію збірки, вона не може бути перезаписана іншою версією тієї самої збірки: нова версія буде доступна паралельно зі старою. Цим вирішують проблему зворотної сумісності – гарантують наявність саме тієї версії динамічної бібліотеки, з розрахунком на яку розроблене застосування.

Висновки

- Особливим видом файлів, які використовують в ОС, є виконувані файли. Їх створюють компіляцією та компонуванням. При цьому необхідно забезпечити відображення символьних імен, що присутні у вихідному коді, на адреси пам'яті, з якими може працювати процесор після завантаження такого файла у пам'ять. Сучасні компонувальники використовують для цього кілька підходів.
- Розрізняють статичне і динамічне компонування. Динамічне компонування, яке набуло широкого використання у сучасних ОС, пов'язане зі збиранням образу процесу у пам'яті із динамічних бібліотек під час його виконання.
- Сучасні формати виконуваних файлів, такі як ELF у Linux і PE у Windows XP, мають подібну структуру. Ця структура відбиває образ процесу, що полегшує його відображення у пам'ять.

Контрольні запитання та завдання

1. На якому етапі роботи компонувальника (на першому проході, після першого проходу, на другому проході, після другого проходу) розробник може бути повідомлений про такі особливі ситуації:
   а) глобальна змінна повторно визначена в декількох об'єктних файлах;
   б) програма не може поміститися у віртуальному адресному просторі;
   в) визначена глобальна змінна, котру жодного разу не використовували;
   г) задане посилання на неіснуючу зовнішню змінну?
2. Чому в сучасних ОС виконувані файли відображають у пам'ять не одним блоком, а секціями?
3. Опишіть, яким чином використання позиційно-незалежного коду спрощує розробку динамічних бібліотек.
4. Як під час компонування виконуваного файла, у якому є звертання до динамічних бібліотек, забезпечити видачу попереджень про нерозв'язані зовнішні посилання?
5. Чому динамічну бібліотеку завжди відображають в адресний простір процесу повністю, хоча з метою економії пам'яті мало б сенс витягати з неї лише ті функції, які використовує процес?
6. Чому в системі, що використовує динамічне компонування, перший виклик функції з динамічної бібліотеки може виконуватися значно довше, ніж наступні?

7. Чому в разі переходу до використання динамічного компонування розмір балансового набору системи може зменшитися? Як зміниться в цьому випадку розмір робочих наборів окремих процесів?
8. Назвіть переваги і недоліки реалізації динамічного компонування в Linux і Windows XP.
9. Розробіть динамічну бібліотеку для Linux і Windows XP, яка міститиме набір функцій із завдання 8 розділу 11. Створіть тестове застосування, що використовує цю бібліотеку.
10. Реалізуйте застосування для Linux і Windows XP, що може бути розширене під час виконання. Інтерфейс модуля розширення задають набором функцій типу `void` без параметрів. Після запуску застосування видає на екран підказку й очікує введення команди з клавіатури. Можливі такі команди: `load ім'я_модуля` (завантаження модуля в пам'ять), `unload ім'я_модуля` (вилучення модуля з пам'яті), `call ім'я_функції` (виклик функції з модуля). Кожен модуль розширення повинен містити код, який виконується під час його завантаження в пам'ять та вилучення з пам'яті. Якщо під час завантаження модуля буде встановлено, що імена його функцій збігаються з іменами функцій, завантажених раніше в складі іншого модуля, треба видавати повідомлення про помилку.
11. На основі результатів завдання 10 з розділу 3 і завдання 10 з розділу 14 реалізуйте командний інтерпретатор з розширеною функціональністю. Кожен модуль розширення містить набір функцій, що реалізують команди інтерпретатора. Після завантаження модуля в пам'ять реалізовані в ньому команди стають доступними для користувача. Як приклад розробіть такі модулі:
    а) `cd_module`, що містить код команд `cd` (зміна поточного каталогу) і `pwd` (відображення імені поточного каталогу);
    б) `exit_module`, що містить код команди `exit`.

Розділ 15

Керування пристроями введення-виведення

- Завдання і організація підсистеми введення-виведення
- Способи виконання операцій введення-виведення
- Введення-виведення у режимі користувача
- Таймери та системний час
- Керування введенням-виведенням в Linux, Unix та Windows XP

У даному розділі розглядатимуться можливості ОС щодо керування пристроями введення-виведення, а саме: загальна організація підсистеми введення-виведення, різні способи виконання зазначених операцій, деякі особливості роботи цієї підсистеми ядра, засоби організації інтерфейсу введення-виведення для прикладних програм, а також особливості функціонування відповідних драйверів для Linux і Windows XP.

15.1. Завдання підсистеми введення-виведення

Основним завданням підсистеми введення-виведення є реалізація доступу до зовнішніх пристроїв із прикладних програм, яка повинна забезпечити:

- ефективність (можливість використання ОС всіх засобів оптимізації, які надає апаратне забезпечення), спільне використання і захист зовнішніх пристроїв за умов багатозадачності;
- універсальність для прикладних програм (ОС має приховувати від прикладних програм відмінності в інтерфейсі апаратного забезпечення, надаючи стандартний інтерфейс доступу до різних пристроїв), при цьому потрібно завжди залишати можливість прямого доступу до пристрою, оминаючи стандартний інтерфейс;
- універсальність для розробників системного програмного забезпечення (драйверів пристроїв), щоб під час розробки драйвера для нового пристрою можна було скористатися наявними напрацюваннями і легко забезпечити інтеграцію цього драйвера у підсистему введення-виведення.

15.1.1. Забезпечення ефективності доступу до пристроїв

Забезпечення ефективності вимагає розв'язання кількох важливих задач.

- Передусім — це коректна взаємодія процесора із контролерами пристроїв. Відомо, що кожен зовнішній пристрій має контролер, який забезпечує керування пристроєм на найнижчому рівні і є фактично спеціалізованим процесором. Після отримання команди від ОС контролер забезпечує її виконання, при цьому пристрій якийсь час не взаємодіє із процесором комп'ютера, тому той може виконувати інші задачі. Виконавши команду, контролер повідомляє системі про завершення операції введення-виведення, генеруючи відповідну подію. Операційній системі в цьому разі потрібно спланувати процесорний час таким чином, щоб драйвери пристроїв могли ефективно реагувати на події контролера та було забезпечене виконання коду процесів користувача.
- Керування пам'яттю під час введення-виведення. Оперативна пам'ять є швидшим ресурсом, ніж зовнішні пристрої, тому ОС може підвищувати ефективність доступу до пристроїв проміжним зберіганням даних у пам'яті (із використанням таких технологій, як кешування і буферизація).

15.1.2. Забезпечення спільного використання зовнішніх пристроїв

Під час спільного використання зовнішніх пристроїв мають виконуватися певні умови.

- ОС повинна мати можливість забезпечувати одночасний доступ кількох процесів до зовнішнього пристрою і розв'язувати можливі конфлікти (тобто необхідна підтримка синхронізації доступу до пристроїв). Деякі пристрої (наприклад, модем або сканер) можна використати тільки одним процесом у конкретний момент часу, тоді як жорсткий диск завжди використовують спільно;
- Слід забезпечити захист пристроїв від несанкціонованого доступу. Такий захист можна організувати або для пристрою як цілого (наприклад, можна відкрити модем для доступу тільки певній групі користувачів), або для деякої підмножини даних пристрою (наприклад, різні файли на жорсткому диску можуть мати різні права доступу).
- У разі спільного використання пристрою треба розподілити операції введення-виведення різних процесів, для того щоб уникнути «накладок» даних одних процесів на дані інших (наприклад, під час спільного використання принтера важливо відрізняти одні задачі від інших і не переходити до друкування результатів наступної задачі до того, як завершилося виведення попередньої).

15.1.3. Універсальність інтерфейсу прикладного програмування

Оскільки пристрої введення-виведення доволі різноманітні, дуже важливо уніфікувати доступ до них із прикладних програм. Для реалізації цієї ідеї підсистема введення-виведення має використовувати набір базових абстракцій, під час застосування яких можна надати доступ до різних зовнішніх пристроїв узагальненим

способом. Для більшості сучасних ОС такою абстракцією є абстракція файла, що відображається як набір байтів, з яким можна працювати за допомогою спеціальних операцій файлового введення-виведення. До таких операцій належать, наприклад, системні виклики відкриття файла `open()`, файлового читання `read()` і записування `write()`, описані у розділі 11. Файл, що відповідає пристрою (його називають файлом пристрою), не відповідає набору даних на диску, а є засобом організації універсального доступу різних компонентів ОС і прикладних програм до деякого пристрою введення-виведення. У цьому розділі зупинимося на цій концепції докладніше.

Зазначимо, що не всі пристрої добре «вписуються» у модель файлового доступу (до подібних пристроїв належить, наприклад, системний таймер). У цьому разі, з одного боку, ОС може надавати унікальний, нестандартний інтерфейс до таких пристроїв, з іншого — стандартного набору файлових операцій може бути недостатньо для використання всіх можливостей пристрою. Для вирішення цієї проблеми можна запропонувати два підходи.

1. Розширити допустимий набір операцій, створивши інтерфейс, що відображає особливості конкретного пристрою. Його будують на основі стандартного файлового інтерфейсу, створюючи операції, характерні для конкретного пристрою. Ці операції, в свою чергу, використовують стандартні виклики, подібні до `read()` і `write()`.
2. Надати прикладним програмам можливість взаємодіяти із драйвером пристрою безпосередньо. Для цього звичайно пропонують універсальний системний виклик (в UNIX його називають `ioctl()`, у Windows XP — `DeviceIoControl()`), параметри якого задають необхідний драйвер, команду, яку потрібно виконати, і дані для неї.

15.1.4. Універсальність інтерфейсу драйверів пристроїв

Як відомо з розділу 2, драйвер пристрою — це програмний модуль, що керує взаємодією ОС із конкретним зовнішнім пристроєм.

Драйвер можна розглядати як транслятор, що отримує на свій вхід команди високого рівня, зумовлені його інтерфейсом із операційною системою, а на виході генерує низькорівневі інструкції, специфічні для апаратного забезпечення, яке він обслуговує. Звичайно на вхід драйвера команди надходять від підсистеми введення-виведення, у більшості ОС їх можна задавати і у прикладних програмах. На практиці драйвери зазвичай записують спеціальні набори бітів у пам'ять контролерів, повідомляючи їм, які дії потрібно виконати. Драйвери практично завжди виконують у режимі ядра. Вони можуть бути завантажені у пам'ять і вивантажені з неї під час виконання.

Набір драйверів, доступних для операційної системи, визначає набір апаратного забезпечення, із яким ця система може працювати. Якщо драйвер для потрібного пристрою відсутній, користувачу залишається лише чекати, поки він не буде створений (альтернативою є самостійна розробка, однак більшість користувачів до розв'язання такого завдання не готові). Часто в цій ситуації користувач змушений перейти до використання іншої ОС. Для того щоб знизити ймовірність такого небажаного розвитку подій, підсистемі введення-виведення потрібно забезпечити

зручний універсальний і добре документований інтерфейс між драйверами й іншими компонентами операційної системи. Наявність такого інтерфейсу і зручного середовища розробки драйверів дає змогу спростити їхнє створення і розширити коло осіб, які можуть цим займатися (розробка драйверів має бути до снаги виробникам відповідного апаратного забезпечення).

Зручність середовища розробки драйверів визначає набір функцій і шаблонів, наданих програмістові. Часто набір засобів, призначений для розробки драйверів під конкретну операційну систему, постачають розробники цієї ОС у вигляді окремого продукту, який називають *DDK* (Driver Development Kit). У нього входять заголовні файли, бібліотеки, можливо, спеціальні версії компіляторів і налагоджувачів, а також документація.

15.2. Організація підсистеми введення-виведення

Загальна структура підсистеми введення-виведення показана на рис. 15.1.

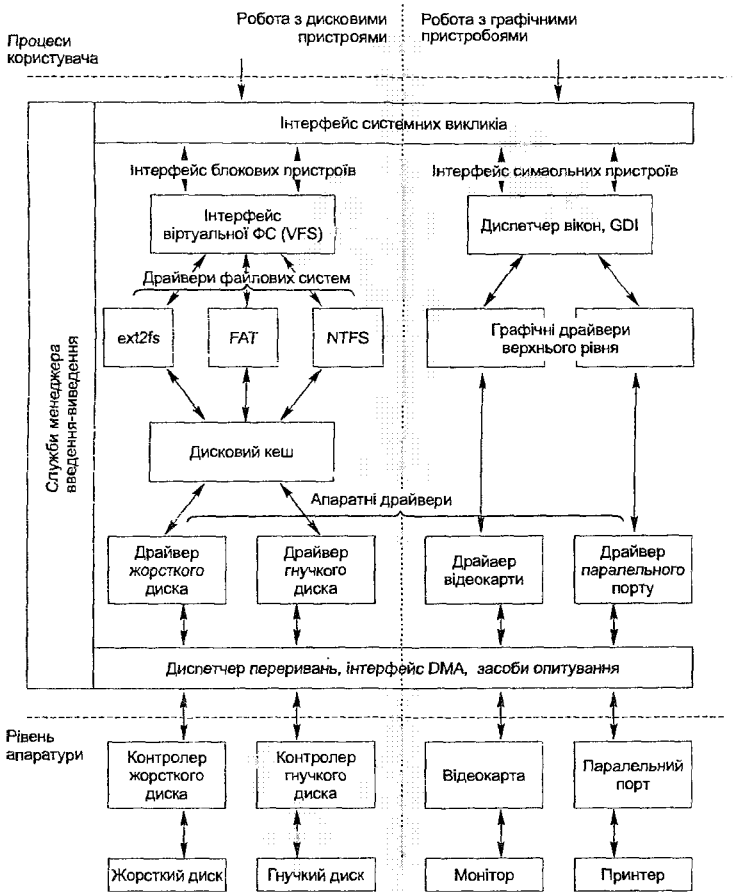

Рис. 15.1. Багаторівнева структура підсистеми введення-виведення [29]

Розглянемо особливості організації такої підсистеми.

- Структура підсистеми розділена на рівні, що дає змогу забезпечити, з одного боку, підтримку всього спектра зовнішніх пристроїв (на нижньому рівні, де розташовані апаратні драйвери), а з іншого — уніфікацію доступу до різних пристроїв (на верхніх рівнях). Зазначимо, що досягти повної ідентичності доступу до всіх можливих пристроїв майже неможливо, тому, крім горизонтальних рівнів, виділяються вертикальні групи пристроїв, взаємодія з якими ґрунтується на загальних принципах. На рис. 15.1 у такі групи об'єднані графічні пристрої та дискові накопичувачі.
- Деякі функції цієї підсистеми не можуть бути реалізовані у драйверах. Це, насамперед, засоби, що забезпечують координацію роботи всіх модулів підтримки введення-виведення у системі, які об'єднуються у *менеджер введення-виведення*. Він, з одного боку, забезпечує доступ прикладних програм до відповідних засобів введення-виведення (наприклад, через інтерфейс файлової системи, реалізований як набір системних викликів), з іншого — середовище функціонування драйверів пристроїв, реалізує загальні служби, такі як буферизація, обмін даними між драйверами тощо.
- Драйвери пристроїв теж розглядають на кількох рівнях. Поряд із низькорівневими апаратними драйверами для керування конкретними пристроями є драйвери вищого рівня. Вони взаємодіють із апаратними драйверами і на підставі отриманих від них даних виконують складніші завдання. Так, над драйвером відеокарти можна розмістити драйвери, які реалізовуватимуть складні графічні примітиви (наприклад, аналогічні рівню GDI у Windows-системах), а над апаратними драйверами дискових пристроїв — драйвери файлових систем. Як наслідок, код драйверів кожного рівня спрощується, кожен із них тепер розв'язує меншу кількість завдань.

15.2.1. Символьні, блокові та мережні драйвери пристроїв

Ще одна важлива класифікація драйверів уперше з'явилася в UNIX і відтоді її широко використовують, оскільки вона добре відображає специфіку різних пристроїв. Пристрої та драйвери відповідно до цієї класифікації розділяються на три категорії: *блокові* або *блок-орієнтовані* (block-oriented, block), *символьні* або *байт-орієнтовані* (character-oriented, character) і *мережні* (network).

- Для блокових пристроїв дані зберігають блоками однакового розміру, при цьому кожен блок має свою адресу, і за допомогою відповідного драйвера до нього можна отримати прямий доступ. Основним блоковим пристроєм є диск.
- Символьні пристрої розглядають дані як потік байтів, при цьому окремий байт адресований бути не може. Прикладами таких пристроїв є модем, клавіатура, миша, принтер тощо. Базовими системними викликами для символьних пристроїв є виклики читання і записування одного байта.
- Окремою категорією є мережні пристрої, які надаються прикладним програмам у вигляді *мережних інтерфейсів* зі своїм набором допустимих операцій, які відображають специфіку мережного введення-виведення (наприклад, ненадійність зв'язку). Деякі ОС реалізують у вигляді мережних драйверів не тільки засоби доступу до пристроїв, але й мережні протоколи.

15.2.2. Відокремлення механізму від політики за допомогою драйверів пристроїв

Концепція драйверів є типовим прикладом розділення механізму і політики в операційних системах. Засобом реалізації механізму доступу до конкретного пристрою є драйвер. Він завжди виконує базові дії із доступу до цього пристрою, не цікавлячись, чому ці дії потрібно виконувати (він вільний від політики). Політику необхідно реалізовувати у програмному забезпеченні вищого рівня, що виконує конкретні операції введення-виведення.

Наприклад, єдине завдання, яке розв'язує драйвер гнучкого диска, – це відображення цього диска у вигляді неперервного масиву блоків даних. Програмне забезпечення вищого рівня надає різні варіанти політики, заснованої на механізмі, реалізованому таким драйвером (наприклад, визначаючи, хто може отримати доступ до цих даних, чи будуть дані доступні прямо або через файлову систему тощо).

15.3. Способи виконання операцій введення-виведення

Зовнішній пристрій взаємодіє із комп'ютерною системою через точку зв'язку, яку називають *портом* (port). Якщо кілька пристроїв з'єднані між собою і можуть обмінюватися повідомленнями відповідно до заздалегідь визначеного протоколу, то кажуть, що вони використовують *шину* (bus).

Як відомо, пристрої зв'язуються із комп'ютером через контролери. Є два базові способи зв'язку із контролером: через *порт введення-виведення* (I/O port) і *відображувану пам'ять* (memory-mapped I/O). У першому випадку дані пересилають за допомогою спеціальних інструкцій, у другому – робота із певною ділянкою пам'яті спричиняє взаємодію із контролером.

Деякі пристрої застосовують обидві технології відразу. Наприклад, графічний контролер використовує набір портів для організації керування і регіон відображуваної пам'яті для зберігання вмісту екрана.

Спілкування із контролером через порт звичайно зводиться до використання чотирьох регістрів. Команди записують у *керуючий регістр* (control), дані – у *регістр виведення* (data-out), інформація про стан контролера може бути зчитана із *регістра статусу* (status), дані від контролера – із *регістра введення* (data-in). Ядро ОС має реєструвати всі порти введення-виведення і діапазони відображуваної пам'яті, а також інформацію про використання пристроєм порту і діапазону.

15.3.1. Опитування пристроїв

Припустимо, що контролер може повідомити, що він зайнятий, увімкнувши біт `busy` регістра статусу. Застосування повідомляє про команду записування вмиканням біта `write` командного регістра, а про те, що є команда – за допомогою біта `cready` того самого регістра. Послідовність кроків базового протоколу взаємодії з контролером (квітування, handshaking), наведено нижче.

1. Застосування у циклі зчитує біт `busy`, поки він не буде вимкнутий.
2. Застосування вмикає біт `write` керуючого регістра і відсилає байт у регістр виведення.

3. Застосування вмикає біт cready.
4. Коли контролер зауважує, що біт cready увімкнутий, то вмикає біт busy.
5. Контролер зчитує значення керуючого регістра і бачить команду write. Після цього він зчитує регістр виведення, отримує із нього байт і передає пристрою.
6. Контролер очищує біти cready і busy, показуючи, що операція завершена.

На першому етапі застосування займається *опитуванням пристрою* (polling), фактично воно перебуває в циклі активного очікування. Одноразове опитування здійснюється дуже швидко, для нього досить трьох інструкцій процесора — читання регістра, виділення біта статусу і переходу за умовою, пов'язаною з цим бітом. Проблеми з'являються, коли опитування потрібно повторювати багаторазово, у цьому разі буде зайвим завантаження процесора. Для деяких пристроїв прийнятним є опитування через фіксований інтервал часу. Наприклад, так можна працювати із дисководом гнучких дисків (цей пристрій є досить повільним, що дає можливість між звертаннями до нього виконувати інші дії).

У більшості інших випадків потрібно організовувати введення-виведення, кероване перериваннями.

15.3.2. Введення-виведення, кероване перериваннями

Про концепцію переривань ішлося у розділах 2 і 3, а тут зупинимося на використанні переривань для організації введення-виведення та особливостях їхньої обробки у драйверах пристроїв.

Базовий механізм переривань дає змогу процесору відповідати на асинхронні події. Така подія може бути згенерована контролером після закінчення введення-виведення або у разі помилки, після чого процесор зберігає стан і переходить до виконання оброблювача переривання, встановленого ОС. Цим знімають необхідність опитування пристрою — система може продовжувати звичайне виконання після початку операції введення-виведення.

Розглянемо деякі додаткові дії, що виникають під час організації обробки переривань у сучасних ОС.

Рівні переривань

Насамперед необхідно мати можливість скасовувати або відкладати обробку переривань під час виконання важливих дій.

Виходячи з цього, переривання поділяють на рівні відповідно до їхнього пріоритету (Interrupt Request Level, IRQL — рівень запиту переривання). Окремі фрагменти коду ОС можуть маскувати переривання, нижчі від певного рівня, скасовуючи їхнє отримання. Виділяють, крім того, немасковані переривання, отримання яких не можна скасувати (апаратний збій пам'яті тощо).

Зазначимо, що за деяких умов переривання вищого рівня можуть переривати виконання оброблювачів нижчого рівня.

Встановлення оброблювачів переривань

Контролер переривань здійснює роботу з перериваннями на апаратному рівні. Він є спеціальною мікросхемою, що дає змогу відсилати сигнал процесору різними лініями. Процесор вибирає оброблювач переривання, на який потрібно перейти, на підставі номера лінії, що нею прийшов сигнал. Її називають лінією запиту переривання або просто *лінією переривання* (IRQ line). У старих архітектурах застосовували-

ся контролери переривань, розраховані на 15-16 ліній переривання і на один процесор, сучасні системи мають спеціальні *розширені програмовані контролери переривань* (Advanced Programmable Interrupt Controllers, APIC), які реалізують багато ліній переривання (наприклад, для стандартного APIC фірми Intel таких ліній 255) і коректно розподіляють переривання між процесорами за умов багатопроцесорних систем.

Ядро ОС зберігає інформацію про всі лінії переривань, доступні у системі. Драйвер пристрою дає запит каналу переривання (IRQ) перед використанням (встановленням оброблювача) і вивільняє після використання. Крім того, різні драйвери можуть спільно використовувати лінії переривань.

Оброблювач переривання може встановлюватися під час ініціалізації драйвера або першого доступу до пристрою (його відкриття). Через обмеженість набору ліній переривання частіше застосовують другий спосіб, у цьому разі, якщо пристрій не використовують, відповідна лінія може бути зайнята іншим драйвером.

Перед тим як встановити оброблювач переривання, драйвер визначає, яку лінію переривання використовуватиме пристрій, який він обслуговує (інакше кажучи, чому дорівнює *номер переривання* для цього пристрою). Є кілька підходів до розв'язання цього завдання.

- Розробник може надати користувачу право самому задати номер цієї лінії. Це — найпростіше вирішення, однак його слід визнати неприйнятним, тому що користувач не зобов'язаний знати номер лінії (для багатьох пристроїв його визначення пов'язане із дослідженням конфігурації перемичок на платі). Усі драйвери у сучасних ОС автоматично визначають номер переривання.
- Драйвер може використати ці номери прямо, оскільки для деяких стандартних пристроїв номер лінії переривання документований і незмінний (або, як для паралельного порту, однозначно залежить від базової адреси відображуваної пам'яті). Але так можна робити далеко не для всіх пристроїв.
- Драйвер може виконати *зондування* (probing) пристрою. Під час зондування драйвер посилає контролеру пристрою запити на генерацію переривань, а потім перевіряє, яка з ліній переривань була активізована. Проте цей підхід є досить незручним, і його вважають застарілим.
- І, нарешті, найкращим є підхід, за якого пристрій сам «повідомляє», який номер переривання він використає. У цьому разі завданням драйвера є просте визначення цього номера, наприклад, читанням регістра статусу одного із портів введення-виведення пристрою або спеціальної ділянки відображуваної пам'яті. Більшість сучасних пристроїв допускають таке визначення конфігурації. До них належать, наприклад, усі пристрої шини PCI (для них це визначено специфікацією, інформацію зберігають у спеціальному просторі конфігурації), а також ISA-пристрої, що підтримують специфікацію Plug and Play.

Особливості реалізації оброблювачів

Оброблювачі переривань — це звичайні послідовності інструкцій процесора; їх можна розробляти як на асемблері, так і мовами програмування високого рівня. В оброблювачах дозволено виконувати більшість операцій за деякими винятками:

- не можна обмінюватися даними із адресним простором режиму користувача, оскільки він не виконується у контексті процесу;
- не можна виконувати жодних дій, здатних спричинити очікування (явно викликати `sleep()`, звертатися до синхронізаційних об'єктів із викликами, які можна

заблокувати, резервувати пам'ять за допомогою операцій, що призводять до сторінкового переривання). Фактично оброблювачі не можуть взаємодіяти із планувальником потоків.

Основні дії оброблювача:

- повідомлення пристрою про те, що переривання оброблене (щоб той міг почати приймати нові переривання);
- читання або записування даних відповідно до специфікації оброблюваного переривання;
- поновлення потоку або кількох потоків, що очікують у черзі, пов'язаній із цим пристроєм (якщо переривання позначає подію, настання якої вони очікували, наприклад прихід нових даних).

Зауважимо, що в деяких випадках для запобігання порушенню послідовності отримання даних оброблювач має на певний час забороняти переривання — або всі, або тільки для його лінії.

Відкладена обробка переривань

Основною вимогою до оброблювачів є ефективність їхньої реалізації. Оброблювач переривання повинен завершувати свою роботу швидко, щоб переривання не залишалися заблокованими надто довго. З іншого боку, часто у відповідь на переривання необхідно виконати досить великий обсяг роботи. Два критерії (швидкість і обсяг роботи) у цьому разі конфліктують один із одним.

Сучасні ОС вирішують цю проблему через поділ коду оброблювача переривання навпіл.

Верхня половина (top half) — це безпосередньо оброблювач переривання, що виконується у відповідь на прихід сигналу відповідною лінією. Зазвичай у верхній половині здійснюють мінімально необхідну обробку (наприклад, повідомляють пристрій про те, що переривання оброблене), після чого вона планує до виконання другу частину.

Нижня половина (bottom half) не виконується негайно у відповідь на переривання, ядро планує її до виконання пізніше, у безпечніший час. Основна відмінність між виконанням обох частин полягає в тому, що під час виконання нижньої половини переривання дозволені, тому вона не впливає на обслуговування інших переривань — ті з них, які виникли після завершення верхньої половини, будуть успішно оброблені. В усьому іншому до коду нижньої половини ставлять ті самі вимоги, що й до коду оброблювачів.

Таку технологію називають *відкладеною обробкою переривань*, вона реалізована в усіх сучасних ОС. У Linux механізми реалізації коду нижньої половини, починаючи із версії 2.4, називають *тасклетами* (tasklets), у Windows XP — *відкладеними викликами процедур* (deferred procedure calls, DPC).

15.3.3. Прямий доступ до пам'яті

Обидва наведені підходи до організації введення-виведення не позбавлені недоліку: вони надто завантажують процесор. Як зазначалося, це є головною проблемою для опитування пристроїв, але введення-виведення на основі переривань теж може мати проблеми, якщо переривання виникатимуть надто часто.

Проблема полягає в тому, що процесор бере участь у кожній операції читання і записування, просто пересилаючи дані від пристрою у пам'ять і назад. Із цією задачею упорався б і простіший пристрій; зазначимо також, що розмір даних, які пересилають за одну операцію, обмежений розрядністю процесора (32 біти, для пересилання наступних 32 біт потрібна нова інструкція процесора). Якщо переривання йдуть часто (наприклад, від жорсткого диска), то час їхньої обробки може бути порівняний із часом, відпущеним для решти робіт. Бажано пересилати дані між пристроєм і пам'яттю більшими блоками і без участі процесора, а його у цей час зайняти більш продуктивними операціями.

Усе це спонукало до розробки *контролерів прямого доступу до пам'яті* (direct memory access, DMA). Такий контролер сам керує пересиланням блоків даних від пристрою безпосередньо у пам'ять, не залучаючи до цього процесора. Блоки даних, які пересилають, завжди набагато більші, ніж розрядність процесора, наприклад вони можуть бути завдовжки 4 Кбайт. Схема введення-виведення при цьому наприклад, буде такою:

- процесор дає команду DMA-контролеру виконати читання блоку від пристрою, разом із командою він відсилає контролеру адресу буфера для введення-виведення (такий буфер має бути у фізичній пам'яті);
- DMA-контролер починає пересилання, процесор у цей час може виконувати інші інструкції;
- після завершення пересилання всього блоку DMA-контролер генерує переривання;
- оброблювач переривання (нижня половина) завершує обробку операції читання, наприклад переміщуючи дані із фізичного буфера у сторінкову пам'ять.

Процесор тут бере участь тільки на початку операції та в кінці — за все інше відповідає контролер прямого доступу до пам'яті.

Завдання ОС під час взаємодії із DMA-контролером досить складні, оскільки такі контролери відчутно відрізняються для різних архітектур і не завжди легко інтегровані із поширеними методами керування пам'яттю. Особливо багато проблем виникає у зв'язку з тим, що буфер для приймання даних від контролера має перебувати у неперервному блоці невивантажуваної фізичної пам'яті. Зазвичай драйвери розміщують буфер для пересилання даних від DMA-пристрою під час ініціалізації і вивільняють після завершення роботи системи; усі операції введення-виведення звертаються до цього буфера. Для драйвера важливо також задання оброблювача переривання від контролера.

15.4. Підсистема введення-виведення ядра

У цьому розділі йтиметься про деякі можливості, які надає підсистема введення-виведення ядра, та їхнє місце в загальній інфраструктурі введення-виведення.

15.4.1. Планування операцій введення-виведення

Планування введення-виведення звичайно реалізоване як середньотермінове планування, особливості якого описані в розділі 4 (розділ 4.2.2, рис. 4.2). Як відомо, з кожним пристроєм пов'язують чергу очікування, під час виконання блокувального

виклику (такого як `read()` або `fcntl()`) потік поміщають у чергу для відповідного пристрою, з якої його звичайно вивільняє оброблювач переривання, як це було описано в розділі 15.3.2. Різним пристроям можуть присвоювати різні пріоритети.

Ще одним прикладом планування введення-виведення є дискове планування, розглянуте в розділі 12.

15.4.2. Буферизація

Найважливішою технологією підвищення ефективності обміну даними між пристроєм і застосуванням або між двома пристроями є буферизація. Для неї виділяють спеціальну ділянку пам'яті, яка зберігає дані під час цього обміну і є буфером. Залежно від того, скільки буферів використовують і де вони перебувають, розрізняють кілька підходів до організації буферизації.

Необхідність реалізації буферизації

Перелічемо причини, які викликають необхідність буферизації.

- Різниця у пропускній здатності різних пристроїв. Наприклад, якщо дані зчитують із модему, а потім зберігають на жорсткому диску, без буферизації процес переходитиме у стан очікування перед кожною операцією отримання даних від модему. Власне кажучи, буферизація потрібна у будь-якій ситуації, коли для читання даних потік має у циклі виконати блокувальну операцію введення-виведення для кожного отримуваного символу; без неї така робота буде вкрай неефективною.
- Різниця в обсязі даних, переданих пристроями або рівнями підсистеми введення-виведення за одну операцію. Типовим прикладом у цьому разі є мережний обмін даними, коли відправник розбиває велике повідомлення на фрагменти, а одержувач у міру отримання поміщає ці фрагменти в буфер (його ще називають *буфером повторного збирання* — reassembly buffer) для того щоб зібрати з них первісне повідомлення. Такий буфер ще більш необхідний, оскільки фрагменти можуть приходити не в порядку відсилання.
- Необхідність підтримки для застосування семантики копіювання (copy semantics). Вона полягає в тому, що інформація, записана на диск процесом, має зберігатися в тому вигляді, у якому вона перебувала у пам'яті в момент записування, незалежно від змін, зроблених після цього.

Способи реалізації буферизації

Розглянемо різні способи реалізації буферизації (рис. 15.2).

Буфер, у який копіюються дані від пристрою, можна організувати в адресному просторі процесу користувача. Хоча такий підхід і дає виграш у продуктивності, його не можна вважати прийнятним, оскільки сторінка із таким буфером може у будь-який момент бути заміщена у пам'яті та скинута на диск. Можна дозволити процесам фіксувати свої сторінки у пам'яті під час кожної операції введення-виведення, але це призводить до невиправданих витрат основної пам'яті.

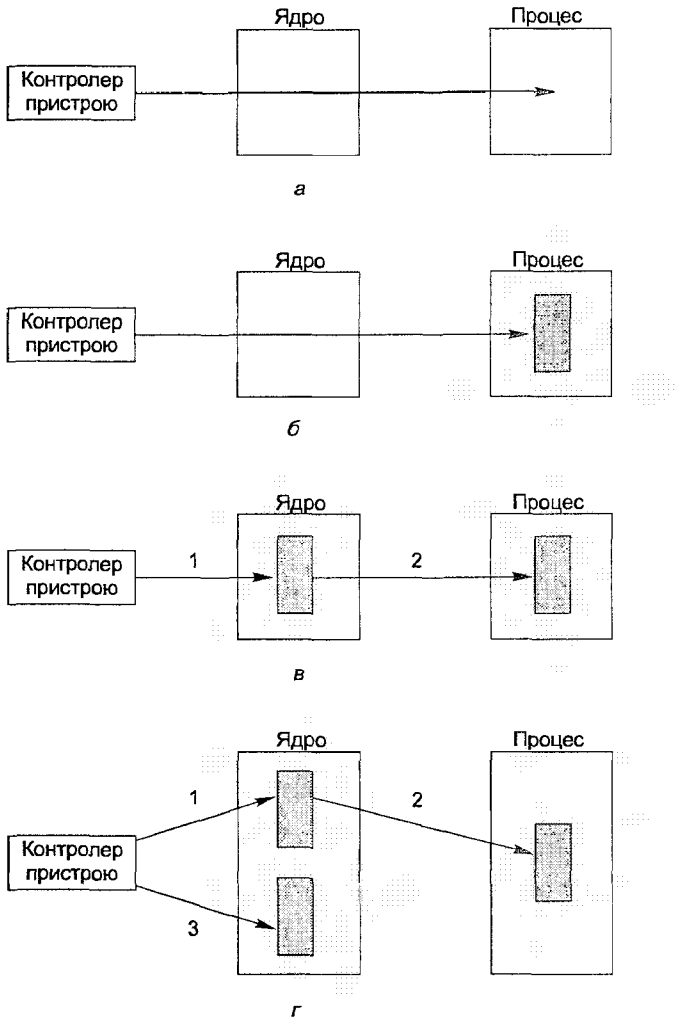

Рис. 15.2. Різні способи буферизації: *а* — введення-виведення без буферизації; *б* — буферизація в просторі користувача; *в* — одинарна буферизація в ядрі; *г* — подвійна буферизація

Першим підходом, який реально використовують на практиці, є одинарна буферизація в ядрі. У цьому разі у ядрі створюють буфер, куди копіюють дані в міру їхнього надходження від пристрою. Коли цей буфер заповнюється, весь його вміст за одну операцію копіюють у буфер, що перебуває у просторі користувача. Аналогічно під час виведення даних їх спочатку копіюють у буфер ядра, після чого вже ядро відповідатиме за їхнє виведення на пристрій. Це дає змогу реалізувати семантику копіювання, оскільки після копіювання даних у буфер ядра інформація із буфера користувача у підсистему введення-виведення гарантовано більше не потрапить — процес може продовжувати свою роботу і використовувати цей буфер для своїх потреб.

Використання одинарного буфера не позбавлене недоліків, головний з яких пов'язаний з тим, що в момент, коли буфер переповнений, нові дані нікуди помістити (а буфер може бути переповнений деякий час, наприклад, поки його зберігають на диску або поки йде завантаження з диска сторінки з буфером користувача). Для вирішення цієї проблеми запропонована технологія подвійної буферизації, за якої у пам'яті ядра створюють два буфери. Коли перший з них заповнений, дані починають надходити в другий, і до моменту, коли заповниться другий, перший уже буде готовий прийняти нові дані і т. д.

Узагальнення цієї схеми на n буферів називають *циклічною буферизацією*, її можна використовувати тоді, коли час збереження буфера перевищує час його заповнення.

Подвійна буферизація дає змогу відокремити виробника даних від їхнього споживача подібно до того, як це робилося для розв'язання задачі виробників-споживачів у розділах 5 і 6.

Буферизація і кешування

Буферизацію варто відрізняти від кешування. Основна відмінність між ними полягає в тому, що буфер може містити єдину наявну копію даних, тоді як кеш за визначенням зберігає у більш швидкій пам'яті копію даних з іншого місця.

З іншого боку, ділянку пам'яті в деяких випадках можна використати і як буфер, і як кеш. Наприклад, якщо після виконання операції введення із використанням буфера надійде запит на таку саму операцію, дані можуть бути отримані із буфера, який при цьому буде частиною кеша. Сукупність таких буферів називають буферним кешем (buffer cache).

Використання буферного кеша дає можливість накопичувати дані для збереження їх на диску великими обсягами за одну операцію, що сприяє підвищенню ефективності роботи підсистеми введення-виведення (не слід забувати при цьому про небезпеку втрати даних у разі вимкнення живлення).

15.4.3. Введення-виведення із розподілом та об'єднанням

Цікавим підходом до оптимізації операцій введення-виведення, пов'язаним із записуванням великих обсягів даних протягом однієї операції, є введення-виведення з розподілом та об'єднанням (scatter-gather I/O). У даному випадку в системі дозволяється використовувати під час введення-виведення набір непов'язаних ділянок пам'яті. При цьому можливі дві дії:

- дані із пристрою відсилають у набір ділянок пам'яті за одну операцію введення (операція розподілу під час введення, scatter);
- усі дані набору ділянок пам'яті відсилають пристрою для виведення за одну операцію (операція об'єднання під час виведення, gather).

Виконання цих дій дає змогу обійтися без додаткових операцій доступу до пристрою, які потрібно було б виконувати, коли всі ділянки пам'яті були використані для введення-виведення по одній.

В UNIX-системах (відповідно до стандарту POSIX) введення із розподілом реалізують системним викликом readv(), виведення з об'єднанням — writev().

```
#include <sys/uio.h>
ssize_t readv(int fd1, const struct iovec *iov, int count);
ssize_t writev(int fd1, const struct iovec *iov, int count);
```

де: fd1 — дескриптор відкритого файла;

iov — масив структур, які задають набір ділянок пам'яті для введення і виведення;

count — кількість структур у масиві iov.

Кожний елемент масиву містить два поля: iov_base, що задає адресу ділянки пам'яті, та iov_len, що задає її довжину. Елемент масиву має бути повністю оброблений (наприклад, заповнений до довжини iov_len за виконання readv()) перед тим як система перейде до обробки наступного.

Виклик readv() повертає загальну кількість зчитаних байтів, а виклик writev() — записаних.

```
int bytes_read;
char buf0[1024], buf1[512];
struct iovec iov[2];
iov[0].iov_base = buf0;
iov[1].iov_base = buf1;
iov[0].iov_len = sizeof(buf1);
iov[1].iov_len = sizeof(buf2);
// infile, outfile – дескриптори відкритих файлів
do {
    // читання у два буфери
    bytes_read = readv(infile, iov, 2);
    // записування із двох буферів
    if (bytes_read > 0) writev(outfile, iov, 2);
} while (bytes_read > 0);
```

Win32 API надає для введення із розподілом функцію ReadFileScatter(), а для виведення з об'єднанням — WriteFileGather().

15.4.4. Спулінг

Спулінг (spooling) — технологія виведення даних із використанням буфера, що працює за принципом FIFO. Такий буфер називають *спулом* (spool) або ділянкою спула (spool area).

Спулінг використовують тоді, коли виведення даних має виконуватися неподільними порціями (*роботами*, jobs). Неподільність робіт полягає в тому, що їхній вміст під час виведення не перемішується (тільки після виведення всіх даних однієї роботи має починатися виведення наступної). Прикладом такого виведення є робота із розподілюваним принтером, коли запити на друкування документів приходять від багатьох процесів у довільному порядку, але друкуватися документи можуть тільки по одному (тут роботою є документ). Іншим прикладом є відсилання електронної пошти (роботами є повідомлення).

Роботи надходять у спул і в ньому вишиковуються у FIFO-чергу (нові роботи додаються у її хвіст). Як тільки пристрій вивільняється, роботу із голови черги

передають пристрою для виведення. Звичайно спулінг пов'язаний із повільними пристроями, тому найчастіше ділянку спула організовують на жорсткому диску, а роботи відображають файлами. Для керування спулом (підтримки черги, пересилання робіт на пристрій) зазвичай використовують фоновий процес або потік ядра. Має бути доступний інтерфейс керування спулом, за допомогою якого можна переглядати вміст черги, вилучати роботи з неї, міняти їхній порядок, тимчасово призупиняти виведення (наприклад, на час обслуговування пристрою).

Зазначимо, що загалом спулінг не можна назвати технологією керування введенням-виведенням режиму ядра — його часто реалізують процеси користувача (такі як демон друкування `lpr` для UNIX).

Менш гнучкою альтернативою спулінгу є надання можливості монопольного захоплення пристрою потоками. Такий монопольний режим може, наприклад, задаватися під час відкриття пристрою, після чого інші потоки не зможуть отримати доступу до нього, поки цей режим не буде знято. Такий підхід вимагає реалізації синхронізації доступу до пристрою. У більшості випадків спулінг є гнучкішим і надійнішим вирішенням, але для таких пристроїв, як записувальний CD-дисковід або сканер, монопольне захоплення є прийнятним варіантом організації доступу.

15.4.5. Обробка помилок

У підсистемі введення-виведення під час роботи виникають різні помилки, які можна віднести до кількох категорій.

- Помилки в програмному коді введення-виведення (доступ до відсутнього пристрою, недопустимі дії із пристроєм тощо). Реакцією на такі помилки звичайно є повернення коду помилки в застосування. Введення-виведення при цьому зазвичай не виконують (найнебезпечнішим видом помилки в цьому разі буде виконання операції із невірним пристроєм, оскільки така операція може відповідати коректній, але зовсім іншій операції для цього пристрою).
- Помилки, викликані апаратними проблемами. Серед них розрізняють:
  - викликані тимчасовими причинами (високе навантаження на мережу, сигнал «зайнято» для модему); для цих помилок звичайною реакцією є повторна спроба виконання введення-виведення;
  - що вимагають втручання користувача (відсутність дискети в дисководі, відсутність паперу у принтері); за такої помилки зазвичай потрібно попросити користувача виконати певні дії;
  - викликані некоректною роботою апаратного забезпечення (збій контролера, дефектні сектори на диску); у цьому разі важливим є надання користувачу (який може виявитися представником служби технічної підтримки фірми-розробника пристрою) якомога більше повної інформації про помилку (зазначимо, що після деяких таких помилок продовження роботи системи може стати неможливим).

У програмному забезпеченні передача докладних відомостей про помилку є головним завданням підсистеми введення-виведення у разі виникнення проблеми. Справді, для більшості системних викликів інтерфейс повідомлень про помилки доволі скромний (цілочисловий код повернення, глобальна змінна `errno`), тому

розробникам драйверів необхідно передбачати додаткові способи отримання інформації про помилку.

15.5. Введення-виведення у режимі користувача

Тут зупинимося на взаємодії підсистеми введення-виведення із процесами режиму користувача та на різних методах організації введення-виведення з режиму користувача.

15.5.1. Синхронне введення-виведення

У більшості випадків введення-виведення на рівні апаратного забезпечення кероване перериваннями, а отже є асинхронним. Однак використати асинхронну обробку даних завжди складніше, ніж синхронну, тому найчастіше введення-виведення в ОС реалізоване у вигляді набору *блокувальних* або *синхронних* системних викликів, подібних до `read()`, `write()` або `fcntl()`. Під час виконання такого виклику поточний потік призупиняють, переміщуючи в чергу очікування для цього пристрою. Після завершення операції введення-виведення і отримання всіх даних від пристрою потік переходить у стан готовності та може продовжити своє виконання.

Однак синхронне введення-виведення підходить не для всіх застосувань. Тут можна навести ті самі приклади, які описані в розділі 3 під час вивчення передумов до внесення паралелізму у застосування (розділ 3.2). Зокрема, воно не підходить для таких категорій програм:

- серверів, що обслуговують багатьох клієнтів (отримавши з'єднання від одного клієнта, потрібно мати можливість відразу обслуговувати й інших);
- застосувань, що працюють із журналом (після виклику функції записування в журнал потрібно продовжити виконання негайно, не очікуючи завершення виведення);
- мультимедійних застосувань (відіславши запит на читання одного кадру, потрібно одночасно показувати інші).

Для вирішення цієї проблеми запропоновано кілька підходів, про які йтиметься нижче.

15.5.2. Багатопотокова організація введення-виведення

Принципи, що лежать в основі першого із можливих підходів до розв'язання проблем синхронного введення-виведення, розглянуто в розділі 3. Цей підхід полягає в тому, що за необхідності виконання асинхронного введення-виведення у застосуванні створюють новий потік, у якому виконуватиметься звичайне, синхронне введення-виведення. При блокуванні цього потоку вихідний потік продовжуватиме своє виконання.

Такий підхід має багато переваг і може бути рекомендований для використання у багатьох видах застосувань. Наведемо приклад розробки багатопотокового сервера за принципом «потік для запиту» (thread per request).

У таких серверах є головний потік, який очікує безпосередніх запитів клієнта на отримання даних (виконуючи синхронну операцію read() або recvfrom() для сокетів). Після отримання кожного запиту головний потік створює новий робочий потік для обробки його запиту, після чого продовжує очікувати подальших запитів. Робочий потік обробляє запит і завершується. Такий підхід застосовують для зв'язку без збереження стану, коли обробка одного запиту не залежить від обробки іншого. Ось псевдокод сервера, що працює за таким принципом:

```
void concurrent_server() {
    for (; ;) {
        read (fd, &request);  // синхронно очікувати запит
        // створити потік для обробки запиту
        create_thread (worker_thread, request);
    }
}

// функція потоку обробки запиту
void worker_thread(request_t request) {
    process_request(request); // обробити запит
}
```

Переваги такого підходу полягають у простоті реалізації та низьких вимогах до ресурсів, недоліки — у недостатній масштабованості (за великої кількості одночасних запитів витрати на створення потоків для кожного із них можуть спричиняти зменшення продуктивності).

У зв'язку з використанням багатопотоковості для організації асинхронного введення-виведення виникають ще деякі проблеми.

- Не завжди варто додавати багатопотоковість в однопотокове застосування тільки тому, що в ньому знадобилося виконати асинхронне введення-виведення.
- Потрібно реалізовувати синхронізацію потоків.
- Може знизитися надійність застосування.
- Кваліфікація програмістів може виявитися недостатньою для реалізації багатопотоковості.

Усе це призводить до значного поширення технологій, альтернативних до цього підходу. Розглянемо їх.

15.5.3. Введення-виведення із повідомленням

Першою технологією, яку можна використати для організації введення-виведення без блокування і яка не вимагає організації багатопотоковості, є *введення-виведення із повідомленням* (notification-driven I/O) [77]. Ця технологія має й інші назви, наприклад мультиплексування введення-виведення (I/O multiplexing).

Загальні принципи введення-виведення із повідомленням

Якщо потрібно в циклі виконати блокувальний виклик (наприклад, read()) для кількох файлових дескрипторів, може трапитися так, що один із викликів заблокує поточний потік у той момент, коли на дескрипторі, який використовується в іншому виклику, з'являться дані. Доцільно організувати одночасне очікування от-

чимання даних із кількох дескрипторів. Це і є основним мотивом розробки даної категорії засобів введення-виведення.

У цьому разі виконання введення-виведення поділяють на кілька етапів.

1. Спеціальний системний виклик (виклик повідомлення) визначає, чи можна виконати синхронне введення-виведення хоча б для одного дескриптора із заданого набору без блокування потоку. У POSIX визначено виклики повідомлення `poll()` і `select()`.
2. Як тільки хоча б один дескриптор із набору стає готовий до введення-виведення без блокування, виклик повідомлення повертає керування; при цьому поточний потік може визначити, для яких саме дескрипторів може бути виконане введення-виведення або які з них змінили свій стан (тобто отримати повідомлення про стан дескрипторів).
3. Потік, що викликає, може тепер у циклі обійти всі дескриптори, визначені внаслідок повідомлення на етапі 2, і виконати введення-виведення для кожного з них, блокування поточного потоку ця операція в загальному випадку не спричинить.

Проілюструємо застосування цієї технології на прикладі реалізації *реактивного сервера* [54]. Такий сервер відповідає на запити клієнтів в одному потоці виконанням у циклі опитування стану набору дескрипторів, визначення тих із них, на які прийшли запити, та подальшого виконання цих запитів.

Введення-виведення із повідомленням про стан дескрипторів

Спочатку реалізуємо реактивний сервер на основі технології *введення-виведення із повідомленням про стан дескрипторів* (state-based notification). Це традиційний підхід, який використовується багато років.

Необхідно виконати такі кроки.

1. Підготувати структуру даних (назвемо її `fdarr`) з описом усіх дескрипторів, стан яких потрібно відстежувати.
2. Передати `fdarr` у системний виклик повідомлення (у POSIX до таких викликів належать уже згадані `select()` і `poll()`). Після виходу із виклику повідомлення `fdarr` міститиме інформацію про стан усіх відстежуваних дескрипторів (які з них готові до виконання введення-виведення без блокування, а які — ні).
3. Для дослідження результатів повідомлення обійти в циклі всі елементи `fdarr` і для кожного із них визначити готовність відповідного дескриптора; якщо він готовий — виконати для нього введення-виведення.

Ось псевдокод реактивного сервера із використанням повідомлення про стан дескрипторів:

```
void reactive_server () {
    // цикл опитування набору дескрипторів fdarr, підготовленого раніше
    for ( ; ; ) {
        select (&fdarr);              // прослуховування набору
        // цикл визначення активних дескрипторів
        for (i=0; i <= count(fdarr); i++) {
            if ( request_is_ready (fdarr[i]) ) {  // якщо був запит
```

```
                read (fdarr[i], &request);  // одержати дані запиту
                process_request (request);  // обслужити клієнта
            }
        }
    }
}
```

Незважаючи на те що такий сервер використовує всього один потік, продуктивність його роботи може бути високою, якщо запити обслуговують достатньо швидко. Сервер працює так:

- `select()` повертає інформацію про ті запити, які потрібно обслужити;
- усі ці запити обслуговують один за одним без затримок на введення-виведення; поки їх обслуговують, надходять нові;
- коли всі запити, про які сповістив минулий виклик `select()`, обслужені, починають нову ітерацію зовнішнього циклу, `select()` викликають знову, і він негайно повертає відомості про всі запити, які надійшли із часу минулого виклику.

Отже, запити потрапляють на обробку групами тим більшими, чим більше приходить запитів. Немає очікування ні під час виклику `select()`, ні під час читання даних із дескриптора. Усе це значно підвищує продуктивність.

Зазначимо, що в циклі обходу `fdarr` було обстежено готовність всіх його елементів до введення-виведення. Виникає запитання: чи завжди обов'язково проводити вичерпне обстеження, чи є можливість отримувати тільки інформацію про готові дескриптори? На жаль, для цього підходу такої можливості немає.

Основною особливістю введення-виведення із повідомленням про стан дескрипторів є те, що в разі його використання не зберігається стан. Кожен виклик повідомлення вимагає передавання всього набору дескрипторів і повертає «миттєвий знімок» стану цих дескрипторів. Це потребує повного обходу цього списку як всередині виклику повідомлення, так і в коді, що його викликав. Такий обхід може серйозно позначитися на продуктивності у разі великої кількості дескрипторів. Таким чином, інформація про неактивні дескриптори даремно копіюватиметься у ядро і назад під час кожного виклику повідомлення.

Введення-виведення із повідомленням про події

Для того щоб підвищити ефективність цієї схеми, запропоновано інший підхід — *введення-виведення із повідомленням про події* (event-based notification) [77].

Основною відмінністю цього підходу є збереження у ядрі інформації про набір дескрипторів, зміна стану яких становить інтерес. Унаслідок цього з'являється можливість повертати інформацію про стан не всіх дескрипторів, а тільки про ті з них, які перейшли у стан готовності з моменту останнього виклику функції повідомлення (тобто, про всі події зміни стану).

Найвідомішими є приклади реалізації такого підходу для FreeBSD (`kqueue`) і для Linux 2.6 (`epoll`).

Виконання введення-виведення в цьому разі зводиться до таких кроків.

1. Спеціальний системний виклик (у Linux — `epoll_create()`) створює структуру даних у ядрі; зазвичай як параметр у такий виклик передають максимальну

кількість дескрипторів, які потрібно контролювати. Таку структуру даних називають прослуховувальним об'єктом.

2. Після створення такого об'єкта для нього потрібно сформувати набір контрольованих дескрипторів, для кожного з них вказують події, які цікавлять потік, що виконував виклик. У Linux це робиться окремим викликом `epoll_ctl()`, один із можливих варіантів виконання якого додає дескриптор у набір.
3. Виклик повідомлення (у Linux – `epoll_wait()`) при цьому повертає інформацію тільки про ті дескриптори, які змінили стан із моменту останнього виклику (а не про всі дескриптори, як для `select()` або `poll()`).

Ось псевдокод реактивного сервера із використанням повідомлення про події:

```
void reactive_server2 () {
    epfd = epoll_create();    // створити прослуховувальний об'єкт
    // додати дескриптори в прослуховувальний об'єкт
    for (i=0; i<=count(fdarr);i++)
        epoll_ctl(epfd, ADD, fdarr[i]);
    // цикл опитування набору дескрипторів у прослуховувальному об'єкті
    for ( ; ; ) {
        epoll_wait (epfd, &active);      // визначення активного набору
        // цикл обслуговування запитів
        for (i=0; i <= count(active); i++) {
            read (active[i], &request);  // одержати дані запиту
            process_request (request);   // обслужити клієнта
        }
    }
}
```

Легко помітити, що внутрішній цикл спростився. Зазначимо також, що використання внутрішньої структури даних позбавляє необхідності обходу всіх дескрипторів усередині `epoll_wait()`.

15.5.4. Асинхронне введення-виведення

Асинхронне введення-виведення реалізоване у деяких UNIX-системах (є стандарт POSIX для таких операцій). Одна з найповніших реалізацій цієї технології доступна також в системах лінії Windows XP, де її називають *введенням-виведенням із перекриттям* (overlapped I/O). Основна ідея тут полягає в тому, що потік, який почав виконувати введення-виведення, не блокують до його завершення.

Асинхронне введення-виведення зводиться до виконання таких дій.

- Потік виконує системний виклик асинхронного введення або виведення, який ставить операцію введення-виведення в чергу і негайно повертає керування.
- Потік продовжує виконання паралельно з операцією введення-виведення.
- Коли операція введення-виведення завершується, потік отримує про це повідомлення.

Операція може бути перервана до свого завершення.

Основним підходом до отримання повідомлення про завершення асинхронного введення-виведення є виконання операції очікування завершення введення-виведення. При цьому потік призупиняють до завершення асинхронної операції.

Після продовження виконання потоку можна отримати інформацію про результат виконання операції. Зазначимо, що тільки на цей час буфер, покажчик на який було передано в операцію асинхронного введення, заповниться зчитаними даними.

Можливий ще один підхід, коли в операцію асинхронного введення-виведення передають адресу процедури зворотного виклику, яка автоматично викликається після завершення виконання операції. Цей підхід складніший у використанні, тому його застосовують рідше.

У разі асинхронного введення-виведення не гарантовано порядок виконання операцій. Якщо, наприклад, потік виконає підряд асинхронні операції читання і записування, немає гарантії, що ОС виконає їх саме в цьому порядку; цілком можливо, що спочатку виконається записування, а потім — читання.

Стандарт POSIX передбачає для виконання асинхронного введення і виведення відповідно виклики `aio_read()` і `aio_write()`, для очікування — `aio_suspend()`, для переривання — `aio_cancel()`, а для отримання результату — `aio_return()`. Ці функції (крім `aio_suspend()`) параметром приймають покажчик на структуру `aiocb` із полями:

- `aio_fildes` — дескриптор файла, для якого здійснено введення-виведення;
- `aio_buf` — покажчик на буфер, у який зчитає дані `aio_read()` і з якого запише дані `aio_write()`;
- `aio_nbytes` — розмір буфера.

Функція `aio_suspend()` має такий вигляд:

```
int aio_suspend(struct aiocb *list[], int cnt, struct timespec *tout);
```

де: `list` — масив покажчиків на структури `aiocb`, можливо, пов'язані із різними операціями введення-виведення (так здійснюють очікування завершення відразу кількох асинхронних операцій);

`cnt` — кількість елементів у масиві `list`;

`tout` — структура, що задає максимальний час очікування (NULL — очікувати нескінченно довго).

Проміжний результат виконання операції можна дістати за допомогою функції `aio_error()`, що повертає значення `EINPROGRESS`, якщо операція ще не була завершена, і нуль — якщо вона завершилась успішно. У цьому разі результат операції (обсяг зчитаних даних для `aio_read()` і записаних — для `aio_write()`) може бути визначений за допомогою виклику функції `aio_return()`.

```
#include <aio.h>
struct aiocb aio_r = { 0 };
char buf[1024];
// заповнення структури aiocb
aio_r.aio_fildes = fopen("myfile.txt", O_RDONLY);
aio_r.aio_buf = buf;
aio_r.aio_nbytes = sizeof(buf);
// починаємо операцію асинхронного читання
aio_read(&aio_r);
// ... тут можна виконувати інші дії
// якщо операція не завершилася - переходимо до очікування
```

```
while (aio_error(&aio_r) == EINPROGRESS) {
    struct aiocb *wait_list = &aio_r;
    aio_suspend(&wait_list,1,NULL);
}
// одержуємо повернуте значення операції
bytes_read = aio_return(&aio_r);
// зчитані дані перебувають в aio_r.aio_buf
```

У Win32 API застосовується трохи інший підхід: для асинхронного введення-виведення можна використати стандартні функції файлового введення і виведення ReadFile() і WriteFile(), якщо в них передається останнім параметром покажчик на спеціальну структуру типу OVERLAPPED. При цьому файл має бути відкритий із увімкнутим прапорцем, що дозволяє асинхронні операції. Для очікування завершення введення-виведення використовують універсальну функцію очікування, наприклад WaitForSingleObject(). У найпростішому випадку вона має очікувати на файловому дескрипторі, для якого було виконане введення або виведення. Для отримання результату треба використати функцію GetOverlappedResult(), для переривання введення-виведення — CancelIo() [32, 50].

Наведемо схему використання асинхронного введення-виведення у Win32 API.

```
// структура - індикатор асинхронної операції
OVERLAPPED ov = { 0 };
// відкрити файл із прапорцем асинхронного введення-виведення
HANDLE fh = CreateFile (..., FILE_FLAG_OVERLAPPED,...);
// асинхронну операцію читання задають передачею структури ov
// останнім параметром
ReadFile (fh, buf, sizeof(buf), &size, &ov);
// ... інші дії, тут buf поки використовувати не можна
WaitForSingleObject (fh, INFINITE);  // очікування завершення операції
// тут можна використати buf
// отримання результату операції
GetOverlappedResult (fh, &ov, &nread, FALSE);
```

15.5.5. Порти завершення введення-виведення

Остання із розглянутих нами технологій — *порт завершення введення-виведення* (I/O completion port) — поєднує багатопотоковість із асинхронним введенням-виведенням для вирішення проблем розробки серверів, що обслуговують велику кількість одночасних запитів [32, 50, 96].

Для кращого розуміння передумов появи цієї технології повернімося до моделі багатопотокового сервера «потік для запиту» із розділу 15.5.1. Як відомо, це вирішення є не досить добре масштабованим через витрати на організацію паралельного виконання великої кількості потоків. Загалом небажано допускати, щоб кількість потоків у системі, які перебувають у стані виконання, набагато перевищувала кількість процесорів.

З іншого боку, запропоновані дотепер альтернативні однопотокові підходи (введення-виведення із повідомленням і асинхронне введення-виведення) не можуть використати переваги багатопроцесорних архітектур. Хотілося б досягти деякого компромісу між великою кількістю потоків і необхідністю використовувати багатопроцесорність.

Одним із варіантів такого компромісу є організація *пула потоків* (thread pool). При цьому в момент запуску серверного застосування заздалегідь створюють набір потоків (пул), кожен із яких готовий обслуговувати запити. Коли приходить запит, перевіряють, чи є в пулі вільні потоки, якщо є, з нього вибирають потік, який починає обслуговувати запит. Після виконання запиту потік повертають у пул. Коли з появою нового запиту вільних потоків у пулі немає, запит поміщають у чергу, і він там очікує, поки не вивільниться потік, що може його обслужити. При цьому можна керувати розміром пула, досягаючи того, щоб кількість активних потоків збігалася із кількістю процесорів у системі.

Для підтримки організації такого пула потоків і було розроблено технологію портів завершення введення-виведення. Такі порти дотепер реалізовані лише у системах лінії Windows XP. Розглянемо особливості їхнього використання.

Особливості використання портів завершення

Спочатку створюють новий об'єкт порту завершення викликом `CreateIoCompletionPort()`. При цьому задають максимальну кількість потоків, пов'язаних із цим портом, які можуть одночасно виконуватися у системі. Позначимо цю величину R_{max}. Рекомендують задавати R_{max} рівним кількості процесорів у системі (якщо задати його рівним нулю, система сама надасть йому цього значення).

```
ph = CreateIoCompletionPort (INVALID_HANDLE_VALUE, 0, 0, Rmax);
```

Після цього із портом пов'язують файлові дескриптори, відкриті для асинхронного введення-виведення. Потоки очікуватимуть його завершення на цих дескрипторах. Для додавання дескриптора в порт також використовують виклик `CreateIoCompletionPort()`, при цьому як параметри в нього повинні бути передані наявний дескриптор порту, файловий дескриптор і унікальний ключ для його однозначного визначення.

```
// додаємо дескриптор з масиву fdarr у порт
CreateIoCompletionPort (fdarr[key], ph, key, Rmax);
```

Після створення порту формують набір (пул) робочих потоків, які обслуговуватимуть запити. Кількість таких потоків має перевищувати R_{max}. Усі потоки пула повинні виконувати один і той самий код, що має приблизно такий вигляд:

```
for ( ; ; ) {
    // очікування на об'єкті порту, заданому дескриптором ph
    GetQueuedCompletionStatus (ph, nbytes, &key, &ov, INFINITE);
    // тут потік є активним
    ReadFile (fdarr[key], request, ...);   // прочитати запит
    process_request (request);             // обслужити клієнта
}
```

Тут виклик `GetQueuedCompletionStatus()` означає опитування порту завершення; саме на ньому очікуватимуть всі потоки пула. Потік стане активним, коли ця функція поверне керування. При цьому вона поверне ключ дескриптора key, заданий під час його додавання. Цей ключ ідентифікує дескриптор, у який прийшов пакет завершення (у прикладі видно використання ключа як індексу у масиві дескрипторів). Активний потік виконуватиме код обробки запиту клієнта.

Клієнти працюють із файловими дескрипторами, пов'язаними із портом, виконуючи для них асинхронне введення-виведення. Завершення його не очікують — за це відповідає порт.

Етапи роботи портів завершення

Розглянемо основні етапи роботи порту завершення.

1. Якщо на жоден дескриптор не прийшло повідомлення про завершення асинхронного введення-виведення, усі потоки очікують на виклик `GetQueuedCompletionStatus()`, перебуваючи у черзі заблокованих потоків. Кажуть, що ці потоки заблоковані на порту завершення. Черга заблокованих потоків фактично і є пулом потоків.
2. Після того як асинхронне введення-виведення для одного із дескрипторів, пов'язаних із портом, завершується (іншими словами, у порт приходить пакет завершення), система перевіряє, скільки активних потоків зараз пов'язані з цим портом (назвемо таку величину R_{cur}).
   - за $R_{cur} < R_{\max}$ один із потоків пула, що очікують на порту завершення, поновлюється і починає обслуговувати запит, стаючи активним потоком; при цьому R_{cur} збільшують на одиницю;
   - за $R_{cur} \geq R_{\max}$ пакет завершення стає у чергу, де й перебуватиме, поки R_{cur} не стане меншим за $R_{\max}$ (що може статися на кроці 3).
3. Коли потік завершує обслуговування запиту, його повертають у пул, блокуючи на порту. Значення R_{cur} зменшують на одиницю і за наявності пакетів, що очікують завершення, один із них починають обробляти. Зазначимо, що пул потоків побудований за принципом LIFO (стека), тобто потік, що починає обробляти пакет завершення, буде потоком, призупиненим останнім (якщо в пул не прийшли нові потоки із початку виконання цього кроку, це буде потік, який щойно повернувся у пул). Це зроблено для оптимізації за малої кількості запитів, оскільки тоді частина потоків взагалі не отримуватиме керування, і їхні ресурси система може вивантажити на диск.
4. У разі заблокування потоку на об'єкті синхронізації під час обслуговування запиту або під час виконання синхронної операції введення-виведення його переміщують у чергу заблокованих потоків, R_{cur} зменшують на одиницю, і пакет, що очікує завершення, починають обробляти аналогічно до кроку 3. Таким чином R_{cur} для порту увесь час підтримують на рівні $R_{\max}$.

   Є можливість явно поміщати пакети завершення у порт за допомогою виклику `PostQueuedCompletionStatus()`. У такий спосіб основний потік застосування може надсилати різні повідомлення робочим потокам.

Розглянемо псевдокод реалізації сервера на основі пула потоків із використанням порту завершення.

```
void port_server() {
    // створення нового порту
    ph = CreateIoCompletionPort(-1,0,0,Rmax);
    // додавання дескрипторів у порт
    for (i=0; i <= count(fdarr); i++)
        CreateIoCompletionPort(fdarr[i], ph, i, Rmax)
```

```
    for (i=0; i <= poolsize; i++)
        create_thread (pool_thread, 0);
}
// функція потоку пула
void pool_thread(request_t request) {
    // код, аналогічний до наведеного раніше
}
```

15.6. Таймери і системний час

Таймери керують пристроями, які передають у систему інформацію про час. Вони відстежують поточний час доби, здійснюють облік витрат процесорного часу, повідомляють процеси про події, що відбуваються через певний проміжок часу тощо. Робота із такими пристроями відрізняється від традиційної моделі введення-виведення, для них використовують окремий набір системних викликів.

15.6.1. Керування системним часом

Апаратний таймер, як уже відомо із розділу 2, – це пристрій, що генерує переривання таймера через певний проміжок часу. Розглянемо, як такий пристрій можна використати для відстеження поточного системного часу.

Таке завдання розв'язують просто: створюють лічильник, який збільшують для кожного переривання таймера. Основною проблемою є розмір цього лічильника, а саме:

- 32-бітне значення не може зберігати достатньо великий проміжок часу (переповнення такого лічильника за частоти переривання таймера 60 Гц настане упродовж двох років);
- 64-бітне значення на 32-бітному процесорі (наприклад, в архітектурі IA-32) оброблятиметься неефективно.

  Для реалізації 32-бітного лічильника звичайно використовують такі підходи.
- Зберігають лише інформацію про секунди, а про долі поточної секунди (мілісекунди, мікросекунди) – окремо. У цьому разі лічильника секунд вистачить для зберігання інформації про 2^{32} с (більш як на 135 років).
- Зберігають інформацію про кількість переривань із моменту останнього завантаження системи, а час останнього завантаження зберігають окремо (як 64-бітне значення). У разі запиту поточного часу значення лічильника і збережений час завантаження додають.

  Якщо поряд із таймером у системі є годинник, значення цього лічильника може час від часу звірятися із показаннями годинника.

Визначення системного часу в Linux

Для визначення системного часу в Linux використовують системний виклик `gettimeofday()`, розглянутий у розділі 7. Зазначимо, що формат часу, який повертає цей виклик, не дуже зручний для використання у застосуваннях (структура `timeval` з полями `tv_sec` – секунди, що пройшли з 1.01.1970, і `tv_usec` – мікросекунди в поточній секунді).

Для перетворення часу у зручний формат використовують функції стандартної бібліотеки мови С, зокрема функцію localtime(). Вона приймає покажчик на поле tv_sec структури timeval і повертає покажчик на структуру tm, поля якої відповідають елементам системного часу: tm_year (рік з 1900), tm_mon (місяць від нуля) і т. д. до tm_sec (секунди).

Розглянемо приклад відображення поточного системного часу у зручному для сприйняття форматі:

```
#include <sys/time.h>
#include <time.h>
struct timeval tv; struct tm *ptm;
gettimeofday(&tv, NULL);
ptm = localtime(&tv.tv_sec);
printf("Зараз %02d/%02d/%d %02d:%02d\n", ptm->tm_mday, ptm->tm_mon+1,
      ptm->tm_year+1900, ptm->tm_hour, ptm->tm_min);
```

Визначення системного часу у Windows XP

Для того щоб дізнатися про поточний системний час у Windows XP, можна використати функцію GetSystemTime():

```
VOID GetSystemTime(LPSYSTEMTIME time);
```

Тут time — покажчик на структуру SYSTEMTIME із полями, що задають елементи системного часу: wYear (рік), wMonth (місяць від одиниці) і т. д. аж до wMilliseconds (мілісекунди).

```
SYSTEMTIME ctime;
GetSystemTime(&ctime);
printf("Зараз %02d/%02d/%d %02d:%02d\n", ctime.wDay, ctime.wMonth,
      ctime.wYear, ctime.wHour, ctime.wMinute);
```

Windows XP постійно коригує системний час за годинником комп'ютера, тому використовувати результат виконання цієї функції для визначення проміжку часу не рекомендовано.

15.6.2. Керування таймерами відкладеного виконання

Іноді процесу потрібно, щоб система сповістила його про закінчення заданого проміжку часу. Наприклад, у програмі, яка реалізує тестування, може бути задано максимальний час відповіді на запитання. Коли цей час минув, процес сповіщається.

Для окремих процесів у системі створюють таймери відкладеного виконання. Їх зазвичай об'єднують у чергу, на початку якої перебуває таймер, що має спрацювати першим (поточний таймер); у ньому зберігають число, яке показує, скільки переривань таймера залишилося до його спрацювання. Кожний наступний таймер у черзі містить число, яке вказує, скільки переривань таймера залишиться до його спрацювання після того, як спрацював попередній.

Для кожного переривання таймера ОС зменшує на одиницю число, котре зберігають у поточному таймері. Коли воно досягає нуля — таймер спрацьовує і його вилучають із черги, а поточним стає наступний за ним.

Аналогічні таймери використовують у ядрі для керування деякими апаратними пристроями. Наприклад, дисковід гнучких дисків не можна використати відразу після ввімкнення двигуна, йому потрібен час для розгону. Для розв'язання цього завдання драйвер диска встановлює таймер після включення двигуна так, щоб він спрацював через час, необхідний для розгону. Після спрацювання такого сторожового таймера (watchdog timer) вважають, що дисковід готовий до роботи.

Інтервальні таймери в Linux

Для забезпечення відкладеного і періодичного виконання коду в Linux використовують інтервальні таймери [24]. Вони керовані системним викликом setitimer() і дають змогу спланувати доставлення сигналу процесу на заданий час у майбутньому.

```
#include <sys/time.h>
int setitimer(int type, const struct itimerval *ptimer,
    struct itimerval *old);
```

де: type — визначає вид таймера (ITIMER_REAL — відлічує системний час і відсилає сигнал SIGALRM, ITIMER_VIRTUAL — відлічує час виконання процесу в режимі користувача і відсилає сигнал SIGVTALRM, ITIMER_PROF — відлічує час виконання процесу в усіх режимах і відсилає сигнал SIGPROF);

ptimer — покажчик на структуру itimerval, що задає параметри таймера (із полями it_value — час, через який буде відіслано сигнал, і it_interval — період відсилання сигналу; обидва ці поля є структурами timeval).

Наведемо приклад використання інтервального таймеру в Linux.

```
struct itimerval mytimer = { 0 };
mytimer.it_value.tv_usec = 1000000;   // початок роботи - через 1 сек.
mytimer.it_interval.tv_usec = 500000; // період - 0.5 сек.
// початок виконання таймера
setitimer(ITIMER_REAL, &mytimer, NULL);
// тут процес буде одержувати сигнал SIGALRM через заданий час
pause();
```

Є спрощений варіант інтерфейсу інтервального таймера — системний виклик alarm(), що змушує систему надіслати процесу сигнал SIGALRM через задану кількість секунд.

```
#include <unistd.h>
alarm(600);   // сигнал надійде через 10 хвилин
```

Таймери очікування у Win32

Аналогами інтервальних таймерів у Win32 є *таймери очікування* (waitable timers) [31]. Такі таймери є синхронізаційними об'єктами, із ними можна використовувати функції очікування. Сигналізація таймерів очікування відбувається через заданий час (можна періодично).

Є два види таймерів очікування: таймери синхронізації і таймери із ручним скиданням. Таймер синхронізації у разі сигналізації переводить у стан готовності до виконання всі потоки, які на ньому очікували, а таймер із ручним скиданням — тільки один потік.

Для створення таймера очікування необхідно використовувати функцію CreateWaitableTimer():

```
HANDLE CreateWaitableTimer(LPSECURITY_ATTRIBUTES psa,
    BOOL manual_reset, LPCTSTR name);
```

Тут manual_reset визначає тип таймера (TRUE — таймер із ручним скиданням). Ця функція повертає дескриптор створеного таймера.

Після створення таймер перебуває в неактивному стані. Для його активізації і керування станом використовують функцію SetWaitableTimer():

```
BOOL SetWaitableTimer(HANDLE ht, const LARGE_INTEGER *endtime,
    LONG period, PTIMERAPCROUTINE pfun, LPVOID pfun_arg, BOOL resume);
```

де: ht — дескриптор таймера;

endtime — час, коли спрацює таймер (за негативного значення — задано відносний інтервал, за позитивного — абсолютний час, вимірюваний у 10^{-7} с);

period — період наступних спрацювань таймера (у мілісекундах), нуль — якщо таймер повинен спрацювати один раз;

pfun – функція користувача, яку викликатимуть у разі спрацювання таймера.

Наведемо приклад використання таймеру очікування у Windows XP.

```
LARGE_INTEGER endtime;
endtime.QuadPart = -10000000;      // сигналізація через 1 сек.
HANDLE ht = CreateWaitableTimer(NULL, FALSE, NULL);
SetWaitableTimer(ht, &endtime, 0, NULL, NULL, TRUE);  // запуск
WaitForSingleObject(ht, INFINITE); // очікування сигналізації
// ... код, виконуваний у разі спрацювання таймера
```

15.7. Керування введенням-виведенням: UNIX і Linux

Цей розділ присвячено реалізації підсистеми введення-виведення в UNIX-системах. Під час розгляду пристроїв спиратимемося на класифікацію (символьні та блокові пристрої), наведену в розділі 15.2. Особлива увага надаватиметься організації уніфікованого доступу до пристроїв через інтерфейс файлової системи.

15.7.1. Інтерфейс файлової системи

Спеціальні файли пристроїв

Для організації уніфікованого доступу до пристроїв введення-виведення важливо вибрати спосіб звертання до них. У цьому розділі розглянемо прийнятий в UNIX-системах підхід *інтерфейсу файлової системи*, за якого пристрої відображаються спеціальними файлами.

У такому разі кожному драйверу пристрою відповідає один або кілька спеціальних файлів пристроїв. Такі файли за традицією поміщаються в каталог /dev, хоча ця вимога не є обов'язковою.

Кожний спеціальний файл пристрою характеризується чотирма атрибутами.

1. *Ім'я файла* використовують для доступу до пристрою із процесів користувача за допомогою файлових операцій. Прикладами імен файлів пристроїв можуть бути /dev/tty0 (перший термінал), /dev/null (спеціальний «порожній» пристрій, все виведення на який зникають), /dev/hda (перший жорсткий диск), /dev/hda1 (перший розділ на цьому диску), /dev/fd0 (дисковід гнучкого диску), /dev/mouse (миша) тощо.
2. *Тип пристрою* дає змогу розрізняти блокові та символьні пристрої. Для символьних тип позначають як 'c', для блокових — як 'b'.
3. *Номер драйвера* (major number) — це ціле число (зазвичай займає 1 байт, хоча може й 2 байти), що разом із типом пристрою однозначно визначає драйвер, який обслуговує цей пристрій. Ядро системи використовує номер драйвера для визначення того, якому драйверу передати керування в разі доступу до відповідного файла пристрою. Зазначимо, що драйвери блокових і символьних пристроїв нумеруються окремо.
4. *Номер пристрою* (minor number) — ціле число, що характеризує конкретний пристрій, для доступу до якого використовують файл. Цей номер передають безпосередньо драйверу під час виконання кожної операції доступу до файла, на його підставі драйвер визначає, який код йому потрібно виконувати.

Наведемо кілька рядків виведення утиліти ls, запущеної в каталозі /dev:

```
brw-rw----   1 shekvl   floppy   2, 0 Aug 30  2001 fd0
brw-rw----   1 root     disk     3, 0 Aug 30  2001 hda
brw-rw----   1 root     disk     3, 1 Aug 30  2001 hda1
lrwxrwxrwx   1 root     root        5 Feb  7  2003 mouse -> psaux
crw-rw-rw-   1 root     root     1, 3 Aug 30  2001 null
crw-------   1 root     root    10, 1 Oct 17 21:09 psaux
crw--w----   1 shekvl   osbook   4, 0 Aug 30  2001 tty0
```

Виділено тип пристрою (перший символ стовпчика атрибутів), номер драйвера і номер пристрою (останні два значення розділені комами), а також ім'я файла. Звідси видно, що /dev/mouse насправді є символічним зв'язком, що вказує на справжній файл пристрою для PS/2-миші (/dev/psaux), і що пристрої, пов'язані з жорстким диском (файли /dev/hda і /dev/hda1), обслуговує той самий драйвер із номером 3.

Розглянемо, як відбувається звертання до драйверів через спеціальні файли. Насамперед, кожен драйвер під час своєї реєстрації у ядрі вказує, який номер драйвера він використовуватиме. Крім того, у коді драйвера мають бути реалізовані файлові операції драйвера. Кожна з них — це реакція на виконання із файлом пристрою стандартних файлових операцій (системних викликів open(), read(), write(), lseek() тощо). У коді кожної операції можна виконати відповідні дії над пристроєм (туди передають номер пристрою, на підставі якого й відбувається вибір пристрою в коді драйвера).

У системі зберігають дві таблиці драйверів: одна — для символьних, інша — для блокових пристроїв. Кожна така таблиця — це масив елементів, проіндексований за номером драйвера. Елементами таблиць драйверів є структури даних, полями кожної з них є покажчики на реалізації файлових операцій відповідного драйвера.

Тип пристрою, номер драйвера і номер пристрою зберігають в індексному дескрипторі відповідного файла. Під час виконання системного виклику для файла пристрою ядро ОС виконує такі дії:

- звертається до індексного дескриптора файла пристрою;
- отримує звідти тип пристрою, номер драйвера і номер пристрою;
- за типом пристрою вибирає потрібну таблицю драйверів;
- за номером драйвера знаходить відповідний елемент таблиці;
- викликає реалізацію файлової операції для драйвера, що відповідає цьому системному виклику, і передає в неї номер пристрою.

Драйвер визначає пристрій за його номером та виконує із ним відповідні дії.

Для створення файлів пристроїв у UNIX використовують утиліту `mknod`, у виклику якої потрібно задати всі чотири характеристики файла:

```
$ mknod /dev/mydevice c 150 1
```

Так створюють файл символьного пристрою, який обслуговуватиме драйвер із номером 150 і передаватиме у його функції номер пристрою 1.

Зазначимо, що файли пристроїв зберігають на диску як звичайні файли, які в будь-який момент можуть бути створені та вилучені. У разі вилучення файла пристрою вилучають лише засіб доступу до драйвера, на сам драйвер це жодним чином не впливає. Якщо згодом файл пристрою створити заново, через нього можна буде знову працювати із пристроєм, звертаючись до драйвера.

Використання для доступу до драйверів інтерфейсу файлової системи дає змогу легко забезпечити захист пристроїв від несанкціонованого доступу — для цього потрібно просто задати для файлів пристроїв відповідні права (такі права розглянемо у розділі 18).

Використання спеціальних файлів

Доступ до спеціального файла розглянемо на прикладі пристрою /dev/random, який можна використати як генератор випадкових чисел:

```
unsigned int randval;
int randfd = open("/dev/random", O_RDONLY);
read(randfd, (char *)&randval, sizeof(randval));
printf("випадкове значення: %u\n", randval);
```

Зазначимо, що, якщо спробувати зчитати великий обсяг даних із цього пристрою, операція читання може бути заблокована доти, поки користувач не виконає додаткових інтерактивних дій із системою (пересуне мишу, введе символи із клавіатури тощо), додаючи джерело випадковості. Якщо така затримка неприйнятна, можна використати пристрій /dev/urandom, операція читання з якого не може бути заблокована.

15.7.2. Передавання параметрів драйверу

Стандартні файлові операції часто не вичерпують усіх дій, які можна робити із пристроями. Для того щоб не доводилося вводити нові системні виклики для додаткових дій, більшість ОС реалізують «таємний хід» — спеціальний системний виклик, що дає змогу передавати драйверу будь-які команди та обмінюватися інформацією в довільному форматі.

Драйвер має реалізувати функцію реакції на такий виклик так само, як реакцію на стандартні файлові виклики. Елемент таблиці драйверів містить поле, призначене для зберігання покажчика на цю функцію.

В UNIX-системах такий виклик називають `ioctl()`.

```
#include <sys/ioctl.h>
int ioctl(int d, int request[, char *argp]);
```

де: `d` — файловий дескриптор, що відповідає відкритому файлові пристрою;

`request` — ціле число, що задає команду драйвера;

`argp` — покажчик на довільну пам'ять, за допомогою якого застосування і драйвер можуть обмінюватися даними будь-якої природи.

Розглянемо приклад використання `ioctl()` в Linux для керування приводом дисководу CD-ROM [24]:

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
int fd = open("/dev/cdrom", O_RDONLY);  // відкрити пристрій
ioctl(fd, CDROMEJECT);                  // виштовхнути CD-ROM
close(fd);
```

Зазначимо, що відповідна команді драйвера константа `CDROMEJECT` оголошена в заголовному файлі `<linux/cdrom.h>`, де перебувають оголошення засобів керування відповідним драйвером.

15.7.3. Структура драйвера

У розділі 15.3.2 вже йшлося про ті частини драйвера, що відповідають за обробку переривань, а в розділі 15.6.1 — про функції, які потрібно реалізувати для забезпечення доступу до цього драйвера через інтерфейс файлової системи. Цей матеріал дає змогу окреслити загальну структуру драйвера в UNIX. Для прикладу розглянемо принципи реалізації драйверів у Linux [94].

Драйвер звичайно складається із коду ініціалізації, реалізації файлових операцій і оброблювачів переривань. Якщо драйвер не використовує введення-виведення, кероване перериваннями, а застосовує опитування пристрою, оброблювачі переривань у ньому можуть не реалізуватись. Далі припускатимемо, що буде використане введення-виведення на базі переривань.

Код ініціалізації виконують під час завантаження ядра (а також під час завантаження у пам'ять модуля із кодом драйвера, якщо він реалізований як модуль ядра). Він складається з однієї функції — `init()`. У ній відбувається реєстрація драйвера у системі (вибір номера драйвера, реєстрація оброблювачів переривань тощо). Зазначимо, що спеціальні файли мають створюватись окремо із режиму користувача утилітою `mknod`.

Код реалізації файлових операцій зводиться до набору функцій, що реалізують реакцію на виконання основних системних викликів файлового введення-виведення. Набір цих функцій розрізняють для блокових і символьних драйверів.

- Для символьних можна реалізувати реакцію на виконання таких викликів, як `open()`, `close()`, `read()`, `write()`, `lseek()`, `select()` (для організації підтримки повідомлення), `mmap()`.

- Для блокових важливим є те, що реалізовані в них функції реакції на операції читання і записування викликають не прямо, а після того, як керування пройшло через буферний кеш (за наявності відповідного блоку в кеші операції його читання або записування можуть зовсім не викликатися). Читання і записування тут за традицією реалізовані у рамках однієї функції, позначеної в UNIX як `strategy()`, а в Linux — `request()`.

Окремо слід виділити реалізацію реакції на універсальний системний виклик `ioctl()`, для якого треба продумати набір команд і необхідні структури даних.

Код оброблювачів переривань, як зазначалося у розділі 15.3.2, складається із верхньої (безпосередньо оброблювача) і нижньої половин. Під час реалізації та реєстрації цих частин потрібно використати визначені засоби підтримки, надані ядром. Верхня половина звичайно планує виконання нижньої і передає їй дані, нижня виконує основну обробку переривання. Виконання нижньої половини може бути відкладене до більш зручного часу.

15.7.4. Виконання операції введення-виведення для пристрою

Наведемо приклад алгоритму обробки запиту введення-виведення в UNIX (не враховуючи особливості функціонування пристрою) [58, 94].

1. Процес користувача виконує системний виклик `read()` для спеціального файла пристрою, перед цим підготувавши буфер у своєму адресному просторі та передавши його адресу в цей системний виклик.
2. Відбувається перехід у привілейований режим для виконання коду драйвера пристрою. Як відомо, кожному процесові у привілейованому режимі доступна вся область даних ядра, тому цей перехід не спричиняє перемикання контексту — система продовжує виконувати той самий процес.
3. На підставі інформації, що зберігається в індексному дескрипторі спеціального файла, викликається функція, зареєстрована як реалізація файлової операції `read()` для відповідного драйвера.
4. Ця функція виконує необхідні підготовчі операції (наприклад, розміщує буфер у пам'яті ядра для збереження даних, отриманих від пристрою), відсилає контролеру пристрою запит на виконання операції читання та переходить у режим очікування (призупиняючи цим процес, що виконав операцію читання); для цього зазвичай використовують функцію `sleep_on()`.
5. Контролер читає дані із пристрою, можливо, заповнюючи буфер, наданий реалізацією реакції на `read()`. Коли читання завершене, він генерує переривання.
6. Апаратне забезпечення активізує верхню половину оброблювача переривання. Код верхньої половини ставить у чергу на виконання код нижньої половини.
7. Код нижньої половини заповнює даними буфер, якщо він не був заповнений контролером, виконує інші необхідні дії для завершення операції введення і поновлює виконання процесу, що очікує.
8. Після поновлення керування знову потрапляє в код реакції на `read()` — цього разу у фрагмент, що слідує за викликом функції `sleep_on()`. Цей код копіює дані із буфера ядра у буфер режиму користувача (код оброблювача цього зробити

не може, оскільки він не має доступу до даних режиму користувача), після чого виконання системного виклику завершується.

9. Керування повертають у процес користувача.

15.8. Керування введенням-виведенням: Windows XP

15.8.1. Основні компоненти підсистеми введення-виведення

Базовим компонентом підсистеми введення-виведення Windows XP є менеджер введення-виведення (I/O Manager). Раніше у розділі 15.2 було розглянуто його основні характеристики, тут зупинимося на тому, як він розв'язує одну із найважливіших задач — передає дані між рівнями підсистеми.

Передавання даних між рівнями підсистеми

Обмін даними між рівнями підсистеми введення-виведення є асинхронним. Більшу частину таких даних подано у вигляді пакетів, які передають від одного компонента підсистеми до іншого, можливо, змінюючи на ходу. Кажуть, що ця підсистема Windows XP є *керованою пакетами* (packet driven). Такі пакети називають *пакетами запитів введення-виведення* (I/O Request Packet, IRP), для стислості називатимемо їх *пакетами IRP*.

Менеджер введення-виведення створює пакет IRP, що відображає операцію введення-виведення, передає покажчик на нього потрібному драйверу і вивільняє пам'ять з-під нього після завершення операції. Драйвер, у свою чергу, отримує такий пакет, виконує визначену в ньому операцію і повертає його назад менеджерові введення-виведення як індикатор завершення операції або для передавання іншому драйверу для подальшої обробки.

Категорії драйверів пристроїв

Windows XP дає змогу використати кілька категорій драйверів режиму ядра. Найбільше поширення останнім часом набули *WDM-драйвери* [35]. На них зупинимося докладніше.

Такі драйвери мають відповідати вимогам стандарту, який називають *Windows Driver Model* (WDM). Його розроблено для драйверів, використовуваних у лінії Windows XP та останніх версіях Consumer Windows (Windows 98/Me). Звичайно для переносу таких драйверів між системами достатньо їх перекомпілювати, а деякі з них сумісні на рівні двійкового коду. Розрізняють три типи WDM-драйверів.

- *Драйвери шини* (bus drivers) керують логічною або фізичною шиною (наприклад, PCI, USB, ISA). Такий драйвер відповідає за виявлення пристроїв, з'єднаних із певною шиною.
- *Функціональні драйвери* (function drivers) керують пристроєм конкретного типу. Драйвери шини надають пристрої функціональним драйверам. Звичайно

тільки функціональний драйвер працює з апаратним забезпеченням пристрою, саме він дає змогу системі використати пристрій.

- *Драйвери-фільтри* (filter drivers) доповнюють або змінюють поведінку інших драйверів.

Насправді жоден драйвер, відповідно до стандарту WDM, не може цілковито відповідати за керування пристроєм, усі вони доповнюють один одного.

Крім WDM-драйверів, у Windows XP підтримують такі категорії драйверів ядра: *файлових систем*, відповідальні за перетворення запитів введення-виведення, що використовують файли, у запити до низькорівневих драйверів пристроїв (наприклад, драйвера жорсткого диска); *відображення* (display drivers) підсистеми Win32, які перетворюють незалежні від пристрою запити GDI-підсистеми в команди графічного адаптера або у прямі операції записування у відеопам'ять; *успадковані*, розроблені для Windows NT.

На доповнення до драйверів ядра Windows XP підтримує драйвери режиму користувача. До них, зокрема, належать драйвери принтерів, які перетворюють незалежні від пристрою запити GDI-підсистеми в команди відповідного принтера і передають ці команди WDM-драйверу (наприклад, драйверу паралельного порту або універсальному драйверу USB-принтера).

Підтримка конкретного пристрою може бути розділена між кількома драйверами. Залежно від рівня цієї підтримки виділяються додаткові категорії драйверів.

- *Клас-драйвери* (class drivers). Реалізують інтерфейс обробки запитів введення-виведення, специфічних для конкретного класу пристроїв, наприклад драйвери дисків або пристроїв CD-ROM.
- *Порт-драйвери* (port drivers). Реалізують інтерфейс обробки запитів введення-виведення, специфічних для певного класу портів введення-виведення; зокрема до цієї категорії належить драйвер підтримки SCSI.
- *Мініпорт-драйвери* (miniport drivers). Керують реальними пристроями (наприклад, SCSI-адаптерами конкретного типу) і реалізують інтерфейс, наданий клас-драйверами і порт-драйверами.

Структура драйвера пристрою

Розглянемо структуру драйвера пристрою [14, 35]. Вона багато в чому подібна до структури, прийнятої в Linux. Можна виділити основні процедури драйвера.

- *Процедура ініціалізації.* Звичайно називається DriverEntry, її виконує менеджер введення-виведення під час завантаження драйвера у систему, і зазвичай вона здійснює глобальну ініціалізацію структур даних драйвера.
- *Процедура додавання пристрою* (add-device routine). Вона має бути реалізована будь-яким драйвером, що підтримує специфікацію Plug and Play. Менеджер Plug and Play викликає цю процедуру, якщо знаходить пристрій, за який відповідає драйвер. У ній звичайно створюють структуру даних, відображувану пристроєм (об'єкт пристрою).
- Набір *процедур диспетчеризації* (dispatch routines), аналогічних функціям файлових операцій у Linux. Ці процедури реалізують дії, допустимі для пристрою (відкриття, закриття, читання, записування тощо). Саме їх викликає менеджер введення-виведення під час виконання запиту.

- *Процедура обробки переривання* (interrupt service routine, ISR) аналогічна коду верхньої половини оброблювача переривання для Linux. Вона є оброблювачем переривання від пристрою, виконується із високим пріоритетом; основне її завдання – запланувати для виконання нижню половину оброблювача (DPC-процедуру).
- *Процедура відкладеної обробки переривання*, DPC-процедура (DPC routine), відповідає коду нижньої половини оброблювача переривання в Linux. Вона виконує більшу частину роботи, пов'язаної з обробкою переривання, після чого сигналізує про необхідність переходу до коду завершення введення-виведення.

Особливості виклику цих процедур під час виконання операції введення-виведення наведено нижче.

15.8.2. Виконання операції введення-виведення для пристрою

Передусім зазначимо, що у Windows XP на внутрішньому рівні всі операції введення-виведення, відображені пакетами IRP, є асинхронними. Будь-яку операцію синхронного введення-виведення відображають у вигляді сукупності асинхронної операції й операції очікування.

Зупинимося на обробці запиту синхронного введення-виведення до одноріневого драйвера. Цей процес зводиться до такого.

1. Запит введення-виведення перехоплює динамічна бібліотека підсистеми (наприклад, підсистема Win32 перехоплює виклик функції `WriteFile()`).
2. Динамічна бібліотека підсистеми викликає внутрішню функцію `NtWriteFile()`, що звертається до менеджера введення-виведення.
3. Менеджер введення-виведення розміщує у пам'яті пакет IRP, що описує запит, і відсилає його відповідному драйверу пристрою викликом функції `IoCallDriver()`.

   Подальші кроки аналогічні до описаних для Linux.
4. Драйвер витягає дані із пакета IRP, передає їх контролеру пристрою і дає йому команду почати введення-виведення.
5. Драйвер викликає функцію очікування, поточний потік при цьому призупиняють. Для асинхронного введення-виведення цей етап не виконують.
6. Коли пристрій завершує операцію, контролер генерує переривання, яке обслуговує драйвер.
7. Драйвер викликає функцію `IoCompleteRequest()` для того щоб повідомити менеджерові введення-виведення про завершення ним обробки запиту, заданого пакетом IRP, після чого виконують код завершення операції.

   На двох останніх етапах зупинимося окремо.

Обслуговування переривань

Принципи обробки переривань введення-виведення у Windows XP майже не відрізняються від розглянутих для Linux. У разі виникнення переривання апаратура викликає оброблювач переривання для даного пристрою. При цьому безпосередній оброблювач (верхня половина) звичайно залишається на рівні переривань

пристрою тільки для того щоб поставити на виконання нижню половину (DPC) і завершитися. Основну роботу здійснює, як і в Linux, нижня половина, що виконується із меншим пріоритетом (на рівні переривань DPC/dispatch). Після завершення обробки драйвер просить менеджера введення-виведення завершити обробку запиту і вилучити із системи пакет IRP.

Завершення запиту введення-виведення

Після завершення виконання функції DPC починається останній етап обробки запиту — завершення введення-виведення (I/O completion).

Таке завершення розрізняють для різних операцій. Звичайно воно зводиться, як і в Linux, до копіювання даних в адресний простір процесу користувача (це може бути буфер введення-виведення або блок статусу операції — структура, задана потоком, що робив виклик).

У разі синхронного введення-виведення адресний простір належить до процесу, що робив виклик, і дані можуть бути записані в нього безпосередньо. Якщо запит був асинхронним, активний потік швидше за все належить до іншого процесу, і потрібно дочекатися, поки адресний простір потрібного процесу не стане доступним (тобто поки не почне виконуватися потік, що викликав операцію). Для цього менеджер введення-виведення планує до виконання спеціальну процедуру, яку називають *APC-процедурою* (від Asynchronous Procedure Call — асинхронний виклик процедури). APC-процедура виконується лише в контексті конкретного потоку, тому очікуватиме, поки цей потік не продовжить своє виконання. Далі вона отримує керування, копіює потрібні дані в адресний простір процесу, що робив виклик, вивільняє пам'ять із-під пакета IRP і переводить файловий дескриптор, для якого виконувалась операція (або інший об'єкт, наприклад, порт завершення введення-виведення) у сигналізований стан, для того щоб потік, який викликав операцію (або будь-який потік, що очікував на цих об'єктах) відновив своє виконання. Після цього введення-виведення вважають завершеним.

Обробка даних багаторівневими драйверами

Підхід із використанням пакетів IRP найзручніший для роботи із багаторівневими драйверами. Особливості обробки в цьому разі опишемо на прикладі виконання запиту до файлової системи, драйвер якої розташований поверх драйвера диска.

Після створення пакета IRP менеджер введення-виведення передає його драйверу верхнього рівня (у нашому випадку — файлової системи). Подальші дії залежать від запиту і реалізації його обробки в цьому драйвері — він може відіслати драйверу нижнього рівня той самий пакет, а може згенерувати і відіслати набір нових пакетів. Ці два підходи розглянемо докладніше.

Повторне використання пакета IRP найчастіше застосовують, коли один запит до драйвера верхнього рівня однозначно транслюється в один запит до драйвера нижнього рівня (наприклад, запит до файлової системи — у запит до драйвера диска на читання одного сектора). Структура пакета IRP розрахована на те, що він буде використаний різними драйверами, розташованими один під одним. Фактично відбувається обробка за принципом стека (пакет із верхнього рівня передають на нижній, обробляють, а потім знову повертають на верхній рівень, як у разі вкладених викликів функцій), тому дані для різних драйверів усередині

пакета IRP організовані у вигляді стека. Окремі позиції в цьому стеку можуть бути заповнені відповідними драйверами на шляху пакета від одного драйвера до іншого. Зазначимо, однак, що розмір пакета під час його переміщень не змінюють — пам'ять для нього відразу виділяють з урахуванням кількості драйверів, через які він має пройти.

З іншого боку, драйвер верхнього рівня може розбити пакет IRP на кілька пов'язаних пакетів, які задають паралельну обробку одного запиту введення-виведення. Наприклад, коли дані для читання розкидані по диску, файлова система може створити набір пакетів, кожен із яких відповідатиме за читання окремого сектора або групи секторів. Усі ці пакети доставляють драйверу пристрою (диска), що обробляє їх по одному, при цьому файлова система відслідковує процес. Після того як усі дії відповідно до пакетів набору виконані, підсистема введення-виведення відновлює первісний пакет і повертає керування процесу або драйверу верхнього рівня.

15.8.3. Передавання параметрів драйверу пристрою

Аналогічно до `ioctl()` в UNIX/Linux, у Win32 API є функція, що безпосередньо викликає команду драйвера пристрою і передає йому необхідні параметри. Це функція `DeviceIoControl()`.

```
BOOL DeviceIoControl(HANDLE hd, DWORD ioc_code, LPVOID in_buf,
  DWORD in_bufsize, LPVOID out_buf, DWORD out_bufsize,
  LPDWORD pbytes_returned, LPOVERLAPPED ov);
```

де: `hd` — дескриптор пристрою, відкритого `CreateFile()`; це може бути том, каталог тощо, імена довільних пристроїв утворюються як `\\.\ім'я_пристрою`;

`ioc_code` — код команди драйвера;

`in_buf` — буфер із вхідними даними для виклику, `in_bufsize` — його довжина;

`out_buf` — буфер з вихідними даними виклику, `out_bufsize` — його довжина;

`pbytes_returned` — покажчик на пам'ять, у яку буде збережено дані, поміщені в `out_buf`.

Наведемо приклад використання `DeviceIoControl()` для реалізації стискання файла на файловій системі NTFS. Щоб стиснути файл, потрібно надіслати драйверу файлової системи команду із кодом `FSCTL_SET_COMPRESSION`. Як вхідні дані при цьому передають формат стискання (заданий цілочисловою змінною зі значенням `COMPRESSION_FORMAT_DEFAULT`). Файл має бути відкритий для читання і записування.

```
unsigned short ctype = COMPRESSION_FORMAT_DEFAULT;
DWORD ret_bytes = 0;
HANDLE fh = CreateFile("myfile.txt", GENERIC_READ|GENERIC_WRITE, ...);
DeviceIoControl(fh, FSCTL_SET_COMPRESSION,
   (LPVOID) &ctype, sizeof(ctype), NULL, 0, &ret_bytes, NULL);
CloseHandle(fh);
```

Зазначимо, що багато функцій керування файловою системою NTFS (зокрема, робота з точками з'єднання) реалізовані через інтерфейс цієї функції [87].

Висновки

- Однією із найважливіших функцій ОС є керування пристроями введення-виведення. Під час його реалізації насамперед важливо безпосередньо реалізувати виконання операцій введення-виведення. Найпоширенішими підходами до розв'язання цього завдання є опитування пристроїв введення-виведення на основі переривань і використання контролерів доступу до пам'яті (DMA).
- Другим важливим завданням є реалізація операцій з організації виконання введення-виведення у ядрі. Основними підходами тут є планування операцій введення-виведення, буферизація і спулінг. Необхідно завжди враховувати можливість виникнення помилок введення-виведення.
- Третім завданням є організація різних засобів введення-виведення для використання в режимі користувача. Сучасні ОС надають різні високоефективні підходи до реалізації таких засобів: синхронне й асинхронне введення-виведення, введення-виведення із повідомленням, порти завершення введення-виведення. Більшість цих засобів розраховані на використання у поєднанні з багатопотоковістю.
- Для реалізації всіх цих можливостей ОС повинна мати драйвери пристроїв, які реалізують базовий набір операцій доступу до пристроїв і надають для використання цих операцій простий у застосуванні універсальний інтерфейс (подібний до інтерфейсу файлової системи в UNIX-сумісних ОС).

Контрольні запитання та завдання

1. Коли краще використовувати опитування завершення введення-виведення, а коли — введення-виведення, кероване перериваннями? Опишіть гібридну стратегію введення-виведення, яка об'єднувала б переваги обох підходів.
2. Для яких із приведених пристроїв є сенс використовувати буферизацію, для яких — спулінг, а для яких — кешування:
   а) миша;
   б) накопичувач на магнітній стрічці (стример);
   в) накопичувач на жорстких магнітних дисках;
   г) графічний адаптер?
3. У сучасних ОС введення-виведення з повідомленням часто дає змогу досягти підтримки більшого числа клієнтів порівняно з іншими підходами. Поясніть, у яких випадках це відбувається і чому.
4. Видозмініть клієнт-серверну систему, розроблену під час виконання завдання 11 з розділу 11. Сервер має реалізовувати архітектуру «потік для запиту». Клієнт для Windows XP повинен використовувати введення-виведення з перекриттям, клієнт для Linux — асинхронне введення-виведення відповідно до POSIX, якщо в системі є його реалізація.
5. Що потрібно робити, якщо у разі використання багатопотокового введення-виведення, введення-виведення з повідомленням або порту завершення введення-виведення надійшов запит, що містить тільки частину даних, необхідних для обслуговування клієнта, і передбачено, що інші дані надійдуть слідом? Пам'ятайте про можливості використання буферизації.

6. Чи можливо, щоб кількість активних потоків для порту завершення введення-виведення перевищило R_{max}? Якщо так, то в якому випадку це трапляється? За яких умов можна сказати, що порт завершення введення-виведення реалізує спулінг?
7. Реалізуйте сервер із завдання 4 цього розділу з використанням порту завершення введення-виведення.
8. Модифікуйте клієнтське застосування з завдання 11 розділу 11, зазначивши максимально допустимий час встановлення з'єднання T_{max}. Виконання застосування має припинятися, якщо з'єднання не було встановлене протягом T_{max}. Використовуйте таймер відкладеного виконання.
9. Назвіть основні відмінності в реалізації та використанні драйверів блокових і символьних пристроїв у Linux.
10. Перелічіть послідовність дій ОС під час виконання операції `write()` для асинхронного пристрою.
11. У Windows XP функціональність підсистеми введення-виведення надана у вигляді об'єктів виконавчої підсистеми. Оцініть переваги і недоліки цього підходу.

Розділ 16

Мережні засоби операційних систем

- Багаторівнева мережна архітектура і мережні протоколи
- Реалізація стека протоколів TCP/IP
- Система імен DNS
- Програмний інтерфейс сокетів Берклі
- Архітектура мережної підтримки Linux та Windows XP
- Програмний інтерфейс Windows Sockets

У цьому розділі буде розглянуто базові принципи і приклади реалізації засобів мережної підтримки в сучасних ОС. Загальні питання організації мережного передавання даних будуть порушені лише тією мірою, якою вони необхідні для викладу основного матеріалу. Докладнішому вивченню цього матеріалу присвячено курс комп'ютерних мереж [13, 21, 28, 42–43].

16.1. Загальні принципи мережної підтримки

Під *мережею* розуміють набір комп'ютерів або апаратних пристроїв (*вузлів*, nodes), пов'язані між собою каналами зв'язку, які можуть передавати інформацію один одному. Мережа має конкретну фізичну структуру (спосіб з'єднання вузлів, топологію), усі вузли підключають до мережі із використанням апаратного забезпечення, яке відповідає цій структурі. Звичайно мережа об'єднує обмежену кількість вузлів.

Під *інтернетом* (з малої літери) розуміють сукупність мереж, які використовують один і той самий набір *мережних протоколів* — правил, що визначають формат даних для пересилання мережею. Фізична структура окремих мереж, які входять до складу інтернету, може різнитися. Такі різнорідні мережі пов'язують одну з одною *маршрутизатори* (routers), які переадресовують пакети з однієї мережі в іншу, залежно від їхньої адреси призначення (маршрутизують їх) і при цьому перетворюють пакети між форматами відповідних мереж. Маршрутизатори підтримують міжмережну взаємодію (internetworking).

Відомий усім Інтернет (з великої літери)— це, фактично, сукупність пов'язаних між собою інтернетів, відкритих для публічного доступу, які використовують визначений набір протоколів (стек протоколів TCP/IP) і охоплюють увесь світ.

16.1.1. Рівні мережної архітектури і мережні сервіси

Функції забезпечення зв'язку між вузлами є досить складними. Для спрощення їхньої реалізації широко використовують багаторівневий підхід – вертикальний розподіл мережних функцій і можливостей. Він дає змогу приховувати складність реалізації функцій зв'язку: кожен рівень приховує від вищих рівнів деталі реалізації своїх функцій та функцій, реалізованих на нижчих рівнях.

У разі використання цього підходу для кожного типу мереж проектують еталонну модель протоколів, що описує функції окремих рівнів і зв'язки між рівнями. Фактично ця модель визначає мережну архітектуру, а рівні є її складовими частинами. Як приклад можна навести мережну архітектуру Інтернету, яку наведено у розділі 16.2.

Мережний сервіс – це набір операцій, які надає рівень мережної архітектури для використання її на вищих рівнях. Сервіси визначено як частину специфікації інтерфейсу рівня.

Розрізняють сервіси, *орієнтовані на з'єднання* (connection-oriented services), і *без з'єднань*, або *дейтаграмні* сервіси (connectionless services).

- Сервіси, орієнтовані на з'єднання, реалізують три фази взаємодії із верхнім рівнем: встановлення з'єднання, передавання даних і розрив з'єднання. При цьому передавання даних на верхніх рівнях здійснюють у вигляді неперервного потоку байтів.
- Дейтаграмні сервіси реалізують пересилання незалежних повідомлень, які можуть переміщатися за своїми маршрутами і приходити у пункт призначення в іншому порядку.

Приклади реалізації сервісів різного типу опишемо нижче. Зазначимо, що реалізацію сервісу на рівні ОС або у вигляді прикладної програми, що надає доступ до деякої системної функціональності через мережу, називають [29] *мережною службою*. Далі вживатимемо цей термін для позначення конкретної програмної реалізації сервісу.

16.1.2. Мережні протоколи

Визначення мережного сервісу для конкретного рівня мережної архітектури описує функціональність цього рівня, але не задає її реалізацію. Реалізацію функціональності для конкретного сервісу визначають мережні протоколи.

Мережний протокол – це набір правил, що задають формат повідомлень, порядок обміну повідомленнями між сторонами та дії, необхідні під час передавання або приймання повідомлень.

Кажуть, що мережний протокол А працює поверх мережного протоколу В (А – протокол вищого рівня, а В – нижчого), коли пакети з інформацією, що відповідають протоколу А, під час передавання мережею розміщені всередині пакетів протоколу В. Процес розміщення одних пакетів усередині інших називають *інкапсуляцією пакетів* (packet encapsulation). У разі приходу пакета за призначенням відповідне програмне забезпечення по черзі «знімає конверти», переглядаючи заголовки пакетів, приймаючи рішення на основі їхнього вмісту і вилучаючи їх. Процес визначення адресата пакета за інформацією із його заголовків називають

демультиплексуванням пакетів (packet demultiplexing). Докладніше інкапсуляцію і демультиплексування пакетів буде розглянуто у розділі 16.2.5.

Мережний протокол надає два інтерфейси.

1. *Однорівневий*, або *інтерфейс протоколу* (peer-to-peer interface) призначений для організації взаємодії із реалізацією протоколу того самого рівня на віддаленому мережному вузлі. Це найважливіший інтерфейс протоколу, що реалізує безпосереднє передавання даних на віддалений вузол. Такий інтерфейс звичайно забезпечують заголовком пакета, який доповнюють реалізацією цього протоколу перед передаванням пакета мережею.
2. *Інтерфейс сервісу* (service interface) призначений для взаємодії із засобами вищого рівня; за його допомогою фактично реалізують мережний сервіс. Інтерфейс сервісу забезпечують правилами інкапсуляції пакетів вищого рівня в пакети цього протоколу.

Набір протоколів різного рівня, що забезпечують реалізацію певної мережної архітектури, називають стеком протоколів (protocol stack) або набором протоколів (protocol suite).

16.2. Реалізація стека протоколів Інтернету

Сукупність протоколів, які лежать в основі сучасного Інтернету, називають *набором протоколів Інтернету* (Internet Protocol Suite, IPS) або *стеком протоколів TCP/IP* за назвою двох основних протоколів [13, 21, 39]. У цьому розділі наведемо основні характеристики такого набору та особливості його реалізації у сучасних ОС.

16.2.1. Рівні мережної архітектури TCP/IP

Мережна архітектура TCP/IP має чотири рівні, яки показані на рис. 16.1. Розглянемо їх знизу вгору.

Канальний рівень (data link layer) відповідає за передавання кадру даних між будь-якими вузлами в мережах із типовою апаратною підтримкою (Ethernet, FDDI тощо) або між двома сусідніми вузлами у будь-яких мережах (SLIP, PPP). При цьому забезпечуються формування пакетів, корекція апаратних помилок, спільне використання каналів. Крім того, на більш низькому рівні він забезпечує передавання бітів фізичними каналами, такими як коаксіальний кабель, кручена пара або оптоволоконний кабель (іноді для опису такої взаємодії виділяють окремий *фізичний рівень* – physical layer).

Перш ніж перейти до наступного рівня, дамо два означення. *Хостом* (host) є вузол мережі, де використовують стек протоколів TCP/IP. *Мережним інтерфейсом* (network interface) є абстракція віртуального пристрою для зв'язку із мережею, яку надає програмне забезпечення канального рівня. Хост може мати декілька мережних інтерфейсів, зазвичай вони відповідають його апаратним мережним пристроям.

На *мережному рівні* (network layer) відбувається передавання пакетів із використанням різних транспортних технологій. Він забезпечує доставлення даних

між мережними інтерфейсами будь-яких хостів у неоднорідній мережі з довільною топологією, але при цьому не бере на себе жодних зобов'язань щодо надійності передавання даних. На цьому рівні реалізована адресація інтерфейсів і маршрутизація пакетів. Основним протоколом цього рівня у стеку TCP/IP є IP (Internet Protocol).

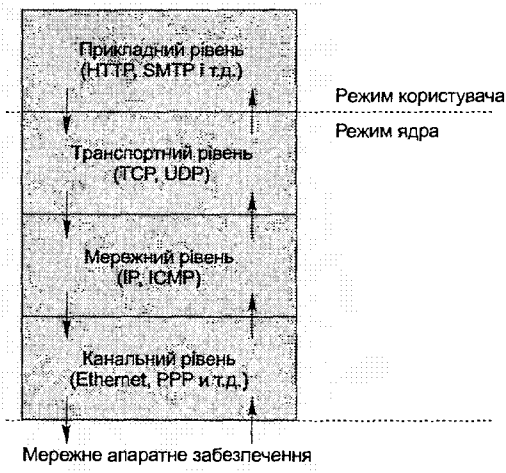

Рис. 16.1. Багаторівнева мережна архітектура TCP/IP

Транспортний рівень (transport layer) реалізує базові функції з організації зв'язку між *процесами*, що виконуються на віддалених хостах. У стеку TCP/IP на цьому рівні функціонують протоколи *TCP* (Transmission Control Protocol) і *UDP* (User Datagram Protocol). TCP забезпечує надійне передавання повідомлень між віддаленими процесами користувача за рахунок утворення віртуальних з'єднань (цей протокол розглядатиметься докладніше у розділі 16.2.4). UDP забезпечує ненадійне передавання прикладних пакетів (подібно до IP), виконуючи винятково функції сполучної ланки між IP і процесами користувача (далі на ньому зупинятися не будемо).

Прикладний рівень (application layer) реалізує набір різноманітних мережних сервісів, наданих кінцевим користувачам і застосуванням. До цього рівня належать протоколи, реалізовані різними мережними застосуваннями (службами), наприклад, HTTP (основа організації Web), SMTP (основа організації пересилання електронної пошти).

Основна відмінність прикладного рівня полягає в тому, що у більшості випадків його підтримка реалізована в режимі користувача (звичайно за це відповідають різні прикладні програми-сервери), а підтримка інших рівнів – у ядрі ОС. Завдання мережної служби прикладного рівня – реалізувати сервіс для кінцевого користувача (пересилання електронної пошти, передавання файлів тощо), який не має інформації про особливості переміщення даних мережею. Інші рівні, навпаки, не мають інформації про особливості застосувань, які обмінюватимуться даними за їхньою допомогою.

16.2.2. Канальний рівень

Реалізація канального рівня звичайно включає драйвер мережного пристрою ОС і апаратний мережний пристрій та приховує від програмного забезпечення верхнього рівня та прикладних програм деталі взаємодії з фізичними каналами, надаючи їм абстракцію мережного інтерфейсу. Передавання даних мережею у програмному забезпеченні верхнього рівня відбувається між мережними інтерфейсами.

Як зазначено вище, кількість мережних інтерфейсів звичайно співвідноситься з кількістю мережних апаратних пристроїв хоста. Крім того, виділяють спеціальний *інтерфейс зворотного зв'язку* (loopback interface); усі дані, передані цьому інтерфейсу, надходять на вхід реалізації стека протоколів того самого хоста.

16.2.3. Мережний рівень

У цьому розділі йтиметься про особливості протоколів мережного рівня.

Протокол IPv4

Протокол IP надає засоби доставлення дейтаграм неоднорідною мережею без встановлення з'єднання. Він реалізує доставлення за заданою адресою, але при цьому надійність, порядок доставлення і відсутність дублікатів не гарантовані. Усі засоби щодо забезпечення цих характеристик реалізуються у протоколах вищого рівня (наприклад, TCP).

Кожний мережний інтерфейс в IP-мережі має унікальну адресу. Такі адреси називають *IP-адресами*. Стандартною версією IP, якою користуються від початку 80-х років XX століття, є IP версії 4 (IPv4), де використовують адреси завдовжки 4 байти. Їх зазвичай записують у крапково-десятковому поданні (чотири десяткові числа, розділені крапками, кожне з яких відображає один байт адреси). Прикладом може бути `194.41.233.1`. Спеціальну адресу зворотного зв'язку `127.0.0.1` (loopback address) присвоюють інтерфейсу зворотного зв'язку і використовують для зв'язку із застосуваннями, запущеними на локальному хості.

Як зазначалося, IP доставляє дейтаграми мережному інтерфейсу. Пошук процесу на відповідному хості забезпечують протоколи транспортного рівня (наприклад, TCP). Пакети цих протоколів інкапсулюють в IP-дейтаграми.

Протокол IPv6

Суттєвим недоліком протоколу IPv4 є незначна довжина IP-адреси. Кількість адрес, які можна відобразити за допомогою 32 біт, є недостатньою з огляду на сучасні темпи росту Інтернету. Сьогодні нові IP-адреси виділяють обмежено.

Для вирішення цієї проблеми було запропоновано нову реалізацію IP-протоколу – *IP версії 6* (IPv6), основною відмінністю якої є довжина адреси – 128 біт (16 байт).

Інші протоколи мережного рівня

Крім IP, на мережному рівні реалізовано й інші протоколи. Для забезпечення мережної діагностики застосовують протокол ICMP (Internet Control Message Protocol), який використовують для передавання повідомлень про помилки під час

пересилання IP-дейтаграм, а також для реалізації найпростішого *луна-протоколу*, що реалізує обмін запитом до хосту і відповіддю на цей запит. ICMP-повідомлення інкапсулюють в IP-дейтаграми.

Більшість сучасних ОС мають утиліту ping, яку використовують для перевірки досяжності віддаленого хоста. Ця утиліта використовує луна-протокол у рамках ICMP.

Підтримка мережного рівня

Засоби підтримки мережного рівня, як зазначалося, є частиною реалізації стека протоколів у ядрі ОС. Головними їхніми завданнями є інкапсуляція повідомлень транспортного рівня (наприклад, TCP) у дейтаграми мережного рівня (наприклад, IP) і передавання підготовлених дейтаграм драйверу мережного пристрою, отримання дейтаграм від драйвера мережного пристрою і демультиплексування повідомлень транспортного рівня, маршрутизація дейтаграм.

16.2.4. Транспортний рівень

Темою цього розділу будуть особливості реалізації протоколів транспортного рівня на прикладі TCP.

Протокол TCP

Пакет з TCP-заголовком називають *TCP-сегментом*. Основні характеристики протоколу TCP [39] такі.

- *Підтримка комунікаційних каналів* між клієнтом і сервером, які називають *з'єднаннями* (connections). TCP-клієнт встановлює з'єднання з конкретним сервером, обмінюється даними з сервером через це з'єднання, після чого розриває його.
- *Забезпечення надійності передавання даних.* Коли дані передають за допомогою TCP, потрібне підтвердження їхнього отримання. Якщо воно не отримане впродовж певного часу, пересилання даних автоматично повторюють, після чого протокол знову очікує підтвердження. Час очікування зростає зі збільшенням кількості спроб. Після певної кількості безуспішних спроб з'єднання розривають. Неповного передавання даних через з'єднання бути не може: або воно надійно пересилає дані, або його розривають.
- *Встановлення послідовності даних* (data sequencing). Для цього кожний сегмент, переданий за цим протоколом, супроводжує номер послідовності (sequence number). Якщо сегменти приходять у невірному порядку, TCP на підставі цих номерів може переставити їх перед тим як передати повідомлення в застосування.
- *Керування потоком даних* (flow control). Протокол TCP повідомляє віддаленому застосуванню, який обсяг даних можливо прийняти від нього у будь-який момент часу. Це значення називають *оголошеним вікном* (advertised window), воно дорівнює обсягу вільного простору у буфері, призначеному для отримання даних. Вікно динамічно змінюється: під час читання застосуванням даних із буфера збільшується, у разі надходження даних мережею — зменшується. Це гарантує, що буфер не може переповнитися. Якщо буфер за-

повнений повністю, розмір вікна зменшують до нуля. Після цього TCP, пересилаючи дані, очікуватиме, поки у буфері не вивільниться місце.

- TCP-з'єднання є *повнодуплексними* (full-duplex). Це означає, що з'єднання у будь-який момент часу можна використати для пересилання даних в обидва боки. TCP відстежує номери послідовностей і розміри вікон для кожного напрямку передавання даних.

Порти

Для встановлення зв'язку між двома процесами на транспортному рівні (за допомогою TCP або UDP) недостатньо наявності IP-адрес (які ідентифікують мережні інтерфейси хостів, а не процеси, що на цих хостах виконуються). Щоб розрізнити процеси, які виконуються на одному хості, використовують концепцію *портів* (ports).

Порти ідентифікують цілочисловими значеннями розміром 2 байти (від 0 до 65 535). Кожний порт унікально ідентифікує процес, запущений на хості: для того щоб TCP-сегмент був доставлений цьому процесові, у його заголовку зазначається цей порт. Процес-сервер звичайно використовує заздалегідь визначений порт, на який можуть вказувати клієнти для зв'язку із цим сервером. Для клієнтів порти зазвичай резервують динамічно (оскільки вони потрібні тільки за наявності з'єднання, щоб сервер міг передавати дані клієнтові).

Для деяких сервісів за замовчуванням зарезервовано конкретні номери портів у діапазоні від 0 до 1023 (відомі порти, well-known ports); наприклад, для протоколу HTTP (веб-серверів) це порт 80, а для протоколу SMTP – 25. В UNIX-системах відомі порти є привілейованими – їх можуть резервувати тільки застосування із підвищеними правами. Відомі порти розподіляються централізовано, подібно до IP-адрес.

Якщо порт зайнятий (зарезервований) деяким процесом, то жодний інший процес на тому самому хості повторно зайняти його не зможе.

Підтримка транспортного рівня

Засоби підтримки транспортного рівня у ядрі призначені для реалізації сервісів, обумовлених цим рівнем. Вони інкапсулюють повідомлення прикладного рівня у сегменти або дейтаграми транспортного рівня, забезпечують необхідні характеристики відповідного протоколу (для TCP до них належать надійність, керування потоком даних тощо), отримують сегменти або дейтаграми від засобів підтримки мережного рівня і демультиплексують їх. Крім того, ці засоби надають інтерфейс системних викликів для використання у прикладних програмах (інтерфейс сокетів, про який ітиметься у розділі 16.4).

16.2.5. Передавання даних стеком протоколів Інтернету

Мережні протоколи стека TCP/IP можна використовувати для зв'язку між рівноправними сторонами, але найчастіше такий зв'язок відбувається за принципом «клієнт-сервер», коли одна сторона (сервер) очікує появи дейтаграм або встановлення з'єднання, а інша (клієнт) відсилає дейтаграми або створює з'єднання [38].

Розглянемо основні етапи процесу обміну даними між клієнтом і сервером із використанням протоколу прикладного рівня, що функціонує в рамках стека протоколів TCP/IP (рис. 16.2). Як приклад такого протоколу візьмемо HTTP, при цьому сторонами, що взаємодіють, будуть веб-браузер (клієнт) і веб-сервер. Припустимо, що локальний комп'ютер зв'язаний з Інтернетом за допомогою мережного пристрою Ethernet. Для простоти вважатимемо, що під час передавання повідомлення не піддають фрагментації. На рис. 16.2 номери етапів позначені цифрами у дужках.

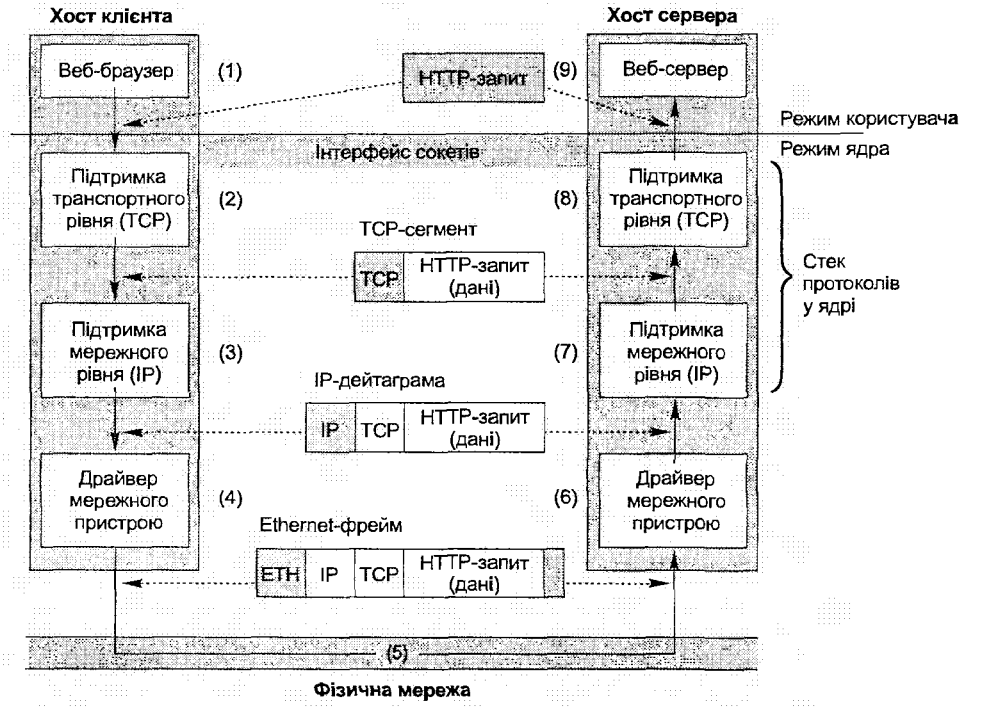

Рис. 16.2. Переміщення пакета у стеку протоколів TCP/IP

1. Застосування-клієнт (веб-браузер) у режимі користувача формує HTTP-запит до веб-сервера. Формат запиту визначений протоколом прикладного рівня (HTTP), зокрема у ньому зберігають шлях до потрібного документа на сервері. Після цього браузер виконує ряд системних викликів (визначених інтерфейсом сокетів, який розглянемо у розділі 16.4). При цьому у ядро ОС передають вміст HTTP-запиту, IP-адресу комп'ютера, на якому запущено веб-сервер, і номер порту, що відповідає цьому серверу.

   Далі перетворення даних пакета відбувається в ядрі.

2. Спочатку повідомлення обробляють засобами підтримки протоколу транспортного рівня (TCP). У результаті його доповнюють TCP-заголовком, що містить номер порту веб-сервера та інформацію, необхідну для надійного пересилання даних (номер послідовності тощо). HTTP-запит інкапсулюється

у TCP-сегмент та стає корисним навантаженням (payload) — даними, які пересилають для обробки в режимі користувача.

3. TCP-сегмент обробляють засобами підтримки протоколу мережного рівня (IP). При цьому він інкапсулюється в IP-дейтаграму (його доповнюють IP-заголовком, що містить IP-адресу віддаленого комп'ютера та іншу інформацію, необхідну для передавання мережею).
4. IP-дейтаграма надходить на рівень драйвера мережного пристрою (Ethernet), який додає до неї інформацію (заголовок і трейлер), необхідну для передавання за допомогою Ethernet-пристрою. Пакет з Ethernet-інформацією називають Ethernet-фрей*мом*. Фрейм передають мережному пристрою, який відсилає його мережею. Фрейм містить адресу призначення у Ethernet, що є адресою мережної карти комп'ютера в тій самий локальній мережі (або іншого пристрою, який може переадресувати пакет далі в напрямку до місця призначення). Апаратне забезпечення Ethernet забезпечує реалізацію передавання даних фізичною мережею у вигляді потоку бітів.

   Дотепер пакет переходив від засобів підтримки протоколів вищого рівня до протоколів нижчого. Кажуть, що пакет опускався у стеку протоколів.
5. Тепер пакет переміщатиметься мережею. При цьому можуть здійснюватися різні його перетворення. Наприклад, коли Ethernet-фрейм доходить до адресата в мережі Ethernet, відповідне програмне або апаратне забезпечення виділяє IP-дейтаграму із фрейму, за IP-заголовком визначає, яким каналом відправляти її далі, інкапсулює дейтаграму відповідно до характеристик цього каналу (наприклад, знову в Ethernet-фрейм) і відсилає її в наступний пункт призначення. На шляху повідомлення може перейти в мережі, зв'язані модемами, і тоді формат зовнішньої оболонки буде змінено (наприклад, у формат протоколів SLIP або PPP), але вміст (IP-дейтаграма) залишиться тим самим.

   Зрештою, пакет доходить до адресата. Його формат залежить від мережного апаратного забезпечення, встановленого на сервері. Якщо сервер теж підключений до мережі за допомогою мережного адаптера Ethernet, він отримає Ethernet-фрейм, подібний до відісланого клієнтом. Далі відбувається декілька етапів демультиплексування пакетів. Кажуть, що пакет піднімається у стеку протоколів.
6. Драйвер мережного пристрою Ethernet виділяє IP-дейтаграму із фрейму і передає її засобам підтримки протоколу IP.
7. Засоби підтримки IP перевіряють IP-адресу в заголовку, і, якщо вона збігається з локальною IP-адресою (тобто IP-дейтаграма дійшла за призначенням), виділяють TCP-сегмент із дейтаграми і передають його засобам підтримки TCP.
8. Засоби підтримки TCP визначають застосування-адресат за номером порту, заданим у TCP-заголовку (це веб-сервер, що очікує запитів від клієнтів). Після цього виділяють HTTP-запит із TCP-сегмента і передають його цьому застосуванню для обробки в режимі користувача.
9. Сервер обробляє HTTP-запит (наприклад, відшукує на локальному диску відповідний документ).

16.3. Система імен DNS

У цьому розділі йтиметься про найважливішу службу прикладного рівня, без якої мережна взаємодія в рамках Інтернету була б фактично неможливою.

16.3.1. Загальна характеристика DNS

Доменна система імен (Domain Name System, DNS) [2] – це розподілена база даних, яку застосування використовують для організації відображення символьних імен хостів (доменних імен) на IP-адреси. За допомогою DNS завжди можна знайти IP-адресу, що відповідає заданому доменному імені. Розподіленість DNS полягає в тому, що немає жодного хосту в Інтернеті, який би мав усю інформацію про це відображення. Кожна група хостів (наприклад, та, що пов'язує всі комп'ютери університету) підтримує свою власну базу даних імен, відкриту для запитів зовнішніх клієнтів та інших серверів. Підтримку бази даних імен здійснюють за допомогою застосування, яке називають DNS-сервером або сервером імен (name server).

Доступ до DNS з прикладної програми здійснюють за допомогою розпізнавача (resolver) – клієнта, який звертається до DNS-серверів для перетворення доменних імен в IP-адреси (цей процес називають *розв'язанням доменних імен* – domain name resolution). Звичайно розпізнавач реалізований як бібліотека, компонована із застосуваннями. Він використовує конфігураційний файл (в UNIX-системах це – /etc/resolv.conf), у якому зазначені IP-адреси локальних серверів імен. Якщо застосування потребує розв'язання доменного імені, код розпізнавача відсилає запит на локальний сервер імен, отримує звідти інформацію про відповідну IP-адресу і повертає її у застосування.

Зазначимо, що і розпізнавач, і сервер імен зазвичай виконуються в режимі користувача (щодо серверів імен це не завжди справедливо: так, у Windows-системах частина реалізації такого сервера виконується в режимі ядра). Стек TCP/IP у ядрі інформацією про DNS не володіє.

В UNIX-системах реалізація сервера імен є окремим продуктом, який називають bind.

16.3.2. Простір імен DNS

Простір імен DNS є ієрархічним (рис. 16.3). Кожний вузол супроводжує символьна позначка. Коренем дерева є вузол із позначкою нульової довжини. Доменне ім'я будь-якого вузла дерева – це список позначок, починаючи із цього вузла (зліва направо) і до кореня, розділених символом «крапка». Наприклад, доменне ім'я виділеного на рис. 16.3 вузла буде «www.kpi.kharkov.ua.». Доменні імена мають бути унікальними.

Доменом (domain) називають піддерево ієрархічного простору імен. Для позначення домену (яке ще називають суфіксом домену) використовують доменне ім'я кореня цього піддерева: так, хост www.kpi.kharkov.ua. належить домену із суфіксом kpi.kharkov.ua., той, у свою чергу, – домену із суфіксом kharkov.ua. і т. д.

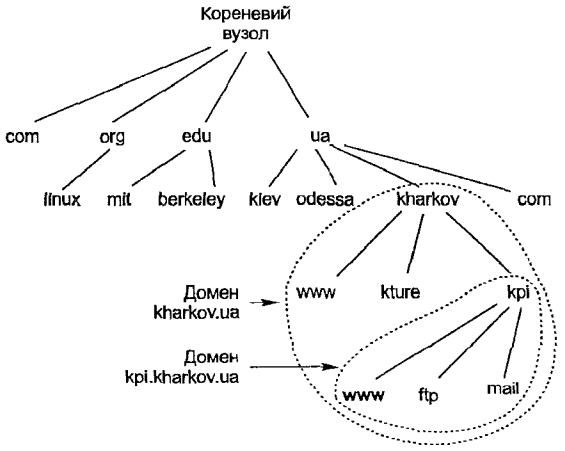

Рис. 16.3. Простір імен DNS

Доменне ім'я, що завершується крапкою, називають *повним доменним іменем* (Fully Qualified Domain Name, FQDN). Якщо крапка наприкінці імені відсутня, вважають, що це ім'я може бути доповнене (до нього може бути доданий суфікс відповідного домену). Такі імена можуть використовуватись у рамках домену. Наприклад, ім'я mail можна використати для позначення хоста всередині домену kpi.kharkov.ua., повне доменне ім'я для цього хоста буде mail.kpi.kharkov.ua.. У застосуваннях крапку наприкінці доменних імен хостів звичайно не ставлять (посилаються на www.kpi.kharkov.ua замість www.kpi.kharkov.ua.), ми теж далі цього не робитимемо.

Серед доменів верхнього рівня (суфікс для яких не містить крапок, окрім кінцевої) виділяють усім відомі com, edu, org тощо, а також домени для країн (ua для України). Є спеціальний домен arpa, який використовують для зворотного перетворення IP-адрес у DNS-імена.

16.3.3. Розподіл відповідальності

Розподіл відповідальності за зони DNS-дерева — найважливіша характеристика доменної системи імен. Немає жодної організації або компанії, яка б керувала відображенням для всіх позначок дерева. Є спеціальна організація (Network Information Center, NIC), що керує доменом верхнього рівня і делегує відповідальність іншим організаціям за інші зони. Зоною називають частину DNS-дерева, що адмініструється окремо. Прикладом зони є домен другого рівня (наприклад, kharkov.ua). Багато організацій розділяють свої зони на менші відповідно до доменів наступного рівня (наприклад, kpi.kharkov.ua, kture.kharkov.ua тощо), аналогічним чином делегуючи відповідальність за них. У цьому разі зоною верхнього рівня вважають частину домена, що не включає виділені в ній зони.

Після делегування відповідальності за зону для неї необхідно встановити кілька серверів імен (як мінімум два — основний і резервний). Під час розміщення в мережі нового хоста інформація про нього повинна заноситься у базу даних

основного сервера відповідної зони. Після цього інформацію автоматично синхронізують між основним і резервним серверами.

16.3.4. Отримання IP-адрес

Якщо сервер імен не має необхідної інформації, він її шукає на інших серверах. Процес отримання такої інформації називають ітеративним запитом (iterative query).

Розглянемо ітеративний запит отримання IP-адреси для імені www.kpi.kharkov.ua. Спочатку локальний сервер зв'язується із кореневим сервером імен (root name server), відповідальним за домен верхнього рівня (.). Станом на 2004 рік в Інтернеті було 13 таких серверів, кожен із них мав бути відомий усім іншим серверам імен. Кореневий сервер імен зберігає інформацію про сервери першого рівня. Отримавши запит на відображення імені, він визначає, що це ім'я належить не до його зони відповідальності, а до домену .ua, і повертає локальному серверу інформацію про адреси та імена всіх серверів відповідної зони. Далі локальний сервер звертається до одного із цих серверів з аналогічним запитом. Той сервер містить інформацію про те, що для зони kharkov.ua є свій сервер імен, у результаті локальний сервер отримує адресу цього сервера. Процес повторюють доти, поки запит не надійде на сервер, відповідальний за домен kpi.kharkov.ua, що може повернути коректну IP-адресу.

16.3.5. Кешування IP-адрес

Кешування IP-адрес дозволяє значно зменшити навантаження на мережу. Коли сервер імен отримує інформацію про відображення (наприклад, IP-адресу, що відповідає доменному імені), він зберігає цю інформацію локально (у спеціальному кеші). Наступний аналогічний запит отримає у відповідь дані з кеша без звертання до інших серверів. Інформацію зберігають у кеші обмежений час. Зазначимо, що, коли для сервера не задана зона, за яку він відповідає, кешування є його єдиним завданням. Це поширений *сервер кешування* (caching-only server).

16.3.6. Типи DNS-ресурсів

Елемент інформації у базі даних DNS називають *ресурсним записом* (Resource Record, RR). Кожний такий запис має клас і тип. Для записів про відображення IP-адрес класом завжди є IN.

Розглянемо деякі типи DNS-ресурсів. Найважливішим із них є *А-запис*, що пов'язує повне доменне ім'я з IP-адресою. Саме на основі таких записів сервери імен повертають інформацію про відображення. У конфігураційному файлі сервера імен bind для домену kpi.kharkov.ua А-запис для www.kpi.kharkov.ua задають так:

```
www     IN   A       144.91.1.21
```

Ще одним типом запису є *CNAME-запис*, що задає *аліас доменного імені*. Одній і тій самий IP-адресі (А-запису) може відповідати кілька таких аліасів. Аліаси ftp.kpi.kharkov.ua і mail.kpi.kharkov.ua задають так:

```
ftp     IN   CNAME   www
mail    IN   CNAME   www
```

16.4. Програмний інтерфейс сокетів Берклі

У розділі 6 наводилися загальні принципи взаємодії між клієнтом і сервером у разі використання сокетів. У цьому розділі опишемо особливості використання інтерфейсу сокетів Берклі (Berkeley Sockets) [38, 47, 75, 106].

Програмний інтерфейс сокетів Берклі – це засіб зв'язку між прикладним рівнем мережної архітектури TCP/IP і транспортним рівнем. Причина такого розташування полягає в тому, що це засіб зв'язку між кодом застосування (який виконують на прикладному рівні) і реалізацією стека протоколу ОС (найвищим рівнем якої є транспортний). Фактично цей інтерфейс є набором системних викликів ОС.

Назва цього API пов'язана з тим, що він уперше був реалізований у 80-х роках XX століття у версії UNIX, розробленій Каліфорнійським університетом у Берклі (BSD UNIX).

16.4.1. Особливості роботи з адресами

Цей розділ буде присвячено особливостям відображення даних про адреси і порти, необхідних для використання сокетів.

Відображення адрес сокетів

Більшість системних викликів роботи із сокетами приймають як параметри покажчики на спеціальні структури даних, що відображають адреси сокетів. У разі використання TCP/IP зі звичайним IP-протоколом (IPv4) адресу задають структурою `sockaddr_in`. Вона визначена у заголовному файлі `<netinet/in.h>` і містить, зокрема, такі поля:

- `sin_family` – для TCP/IP має дорівнювати константі `AF_INET`;
- `sin_port` – цілочисловий номер порту для TCP або UDP;
- `sin_addr` – відображення IP-адреси (структура `in_addr` з одним полем `s_addr`).

Реалізуючі системні виклики роботи із сокетами, необхідно враховувати можливість використання різних протоколів, а отже, передавання в них різних структур, що відображають адреси. Під час розробки API сокетів Берклі було ухвалено рішення використати як універсальний тип відображення адрес покажчик на спеціальну структуру `sockaddr`, визначену у `<sys/socket.h>`.

Наприклад, прототип системного виклику `bind()`, що пов'язує сокет з адресою, має такий вигляд:

```
int bind(int, struct sockaddr *, socklen_t);
```

На практиці адреса задається так: визначають структуру типу `sockaddr_in` і під час виклику покажчик на неї перетворюють до `sockaddr *`:

```
struct sockaddr_in my_addr;
// ... заповнення структури my_addr, див. далі
bind(sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr));
```

Безпосередньо структуру `sockaddr` у застосуваннях не використовують. Якщо виклик повернув покажчик на неї, перед використанням у застосуванні його

потрібно привести до покажчика на структуру для конкретного протоколу (наприклад, на sockaddr_in).

Порядок байтів

Історично так склалося, що відображення у пам'яті даних, для яких виділяють більш як один байт (наприклад, двобайтового цілого, що відповідає у C типу short), відрізняється для різних комп'ютерних архітектур. У деяких із них старший байт розташовують у пам'яті перед молодшим (такий порядок називають зворотним – big-endian byte order), а в інших – молодший перед старшим (це прямий порядок – little-endian byte order). Наприклад, архітектура IA-32 використовує прямий порядок байтів, а комп'ютери Sun Sparcstation та Power PC – зворотний.

Проблема полягає в тому, що деякі елементи пакетів (керуючі дані), якими обмінюються комп'ютери для організації передавання інформації, займають у пам'яті більш як один байт. Так, відображення порту займає 2 байти, а IP-адреси – 4 байти. Якщо такий блок інформації буде передано комп'ютеру з іншим порядком байтів, він не зможе коректно його інтерпретувати.

Для вирішення цієї проблеми реалізації стека протоколів на клієнті та на сервері потрібно під час передавання даних мережею узгоджено використовувати єдиний порядок байтів. Такий порядок, який називають мережним порядком байтів (network byte order), визначає мережний протокол. Фактично, це зворотний порядок. Відповідно, порядок байтів, прийнятий для локального комп'ютера, називають порядком байтів хоста (host byte order). Він може бути як прямим, так і зворотним.

Програміст має перетворювати керуючі дані в мережний порядок байтів перед передаванням інформації із мережі та робити зворотне перетворення після її отримання. Це зокрема стосується IP-адрес і номерів портів. Для реалізації перетворення використовують функції htonl() і htons() (перетворення до мережного порядку), ntohl() і ntohs() (перетворення до порядку хоста):

```
#include <netinet/in.h>
uint16_t htons(uint16_t hostval); // htonl() - для 32-бітних даних
uint16_t ntohs(uint16_t netval);  // ntohl() - для 32-бітних даних
```

Далі наведемо приклади використання цих функцій.

Робота з IP-адресами

У протоколі IPv4 IP-адреса є цілим значенням розміром 4 байти. Для того щоб перетворити крапково-десяткове подання такої адреси (n.n.n.n) у ціле число, слід використати функцію inet_aton():

```
#include <arpa/inet.h>
int inet_aton(const char *c_ip, struct in_addr *s_ip);
```

де: c_ip – рядкове відображення IP-адреси: "192.168.10.28";

s_ip – структура in_addr, що міститиме цілочислове відображення адреси, розташоване в пам'яті відповідно до мережного порядку байтів (поле s_addr); цю структуру заповнюють усередині виклику.

У разі успішного перетворення ця функція повертає ненульове значення, у разі помилки – нуль.

Розглянемо приклад заповнення структури, що задає адресу сокета:

```
struct sockaddr_in my_addr = { 0 };  // структура має бути обнулена
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(7500);      // задання порту
inet_aton("192.168.10.28", &(my_addr.sin_addr));  // задання адреси
```

Для обнулення структур часто використовують функцію bzero():

```
#include <strings.h>
bzero(&my_addr, sizeof(my_addr));
// заповнення my_addr
```

Зворотне перетворення IP-адреси (із цілого числа у крапково-десяткове подання) здійснюють функцією inet_ntoa():

```
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr s_ip);
```

Ця функція розміщає у статичному буфері рядок із крапково-десятковим поданням IP-адреси, заданої структурою s_ip, і повертає покажчик на цей буфер:

```
printf("IP-адреса: %s\n", inet_ntoa(my_addr.s_addr));
```

Зазначимо, що наступні спроби виклику функції стиратимуть статичний буфер, тому для подальшого використання його вміст потрібно копіювати в окремий буфер пам'яті.

16.4.2. Створення сокетів

Перший етап, який необхідно виконати на клієнті та сервері, — створити дескриптор сокета за допомогою системного виклику socket():

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

де: family — комунікаційний домен (AF_INET — домен Інтернету на основі IPv4, AF_UNIX — домен UNIX);

type — тип сокета (SOCK_STREAM — потоковий сокет, SOCK_DGRAM — дейтаграмний сокет, SOCK_RAW — *сокет прямого доступу* (raw socket), призначений для роботи із протоколом мережного рівня, наприклад IP);

protocol — тип протоколу (звичайно 0; це означає, що протокол вибирають автоматично, виходячи з домену і типу сокета; наприклад, потокові сокети домену Інтернет використовують TCP, а дейтаграмні — UDP).

Ось приклад використання socket():

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Виклик socket() повертає цілочисловий дескриптор сокета або –1, якщо сталася помилка (при цьому, як і для інших викликів, змінна errno міститиме код помилки). Цей сокет можна використовувати для різних цілей: організації очікування з'єднань сервером, задання з'єднання клієнтом, отримання інформації про мережну конфігурацію хоста (в останньому випадку він може бути навіть не пов'язаний з адресою).

Для системних викликів інтерфейсу сокетів Берклі далі за замовчуванням передбачатиметься використання заголовного файла <sys/socket.h>.

16.4.3. Робота з потоковими сокетами

Тут розглядатимуться головні системні виклики, які використовують під час розробки серверів та клієнтів з використанням потокових сокетів на основі протоколу TCP (рис. 16.4). Це – основний вид сокетів, що найчастіше трапляється в реальних застосуваннях.

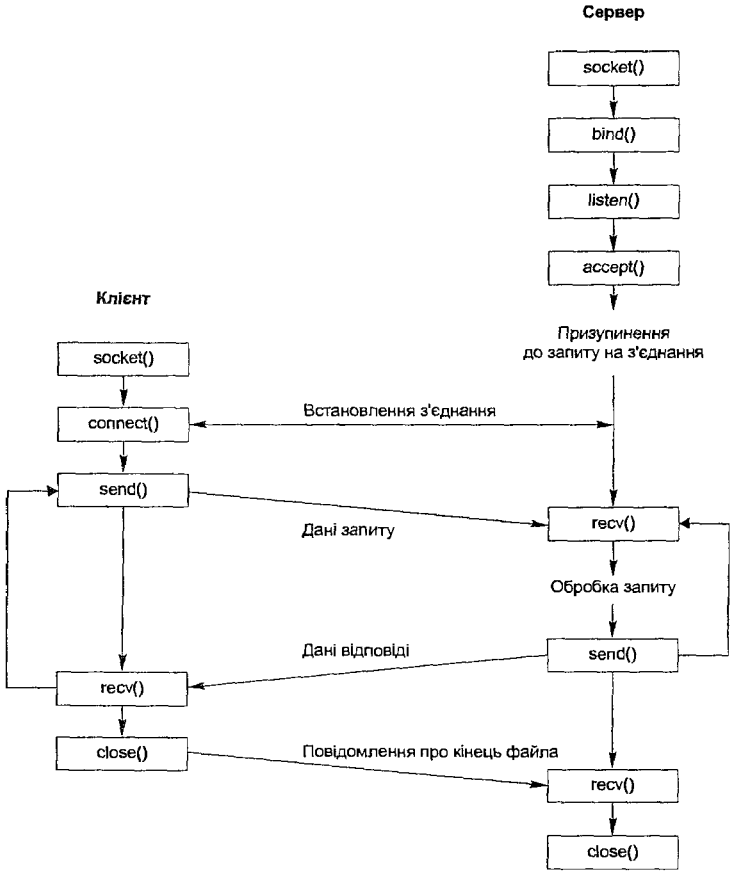

Рис. 16.4. Обмін даними на основі потокових сокетів (за [38])

Зв'язування сокета з адресою

Сокет, дескриптор якого повернутий викликом `socket()`, не пов'язаний із конкретною адресою і недоступний для клієнтів. Для пов'язування сокета з адресою необхідно використати системний виклик `bind()`:

```
int bind(int sockfd, const struct sockaddr *pmy_addr, socklen_t alen);
```

де: `sockfd` – дескриптор сокета, створений за допомогою `socket()`;

pmy_addr — покажчик на заздалегідь заповнену структуру, що задає адресу сокета (наприклад, sockaddr_in);

alen — розмір структури, на яку вказує pmy_addr.

Цей виклик повертає –1, якщо виникла помилка.

Під час виконання bind() застосуванням-сервером необхідно задати наперед відомий порт, через який клієнти зв'язуватимуться з ним. Можна також вказати конкретну IP-адресу (при цьому допускатимуться лише з'єднання, для яких вона зазначена як адреса призначення), але найчастіше достатньо взяти довільну адресу локального хоста, скориставшись константою INADDR_ANY:

```
struct sockaddr_in my_addr = { 0 };
int listenfd = socket(...);
// ... задача my_addr.sin_family і my_addr.sin_port
my_addr.sin_addr.s_addr = INADDR_ANY;
bind(listenfd, (struct sockaddr *)&my_addr, sizeof(my_addr));
```

Найпоширенішою помилкою під час виклику bind() є помилка з кодом EADDRINUSE, яка свідчить про те, що цю комбінацію «IP-адреса — номер порту» вже використовує інший процес. Часто це означає повторну спробу запуску того самого сервера.

```
if (bind(listenfd, ...) == -1 && errno == EADDRINUSE)
    { printf("помилка, адресу вже використовують\n"); exit(-1); }
```

Щоб досягти гнучкості у вирішенні цієї проблеми, рекомендують дозволяти користувачам налаштовувати номер порту (задавати його у конфігураційному файлі, командному рядку тощо).

Відкриття сокета для прослуховування

За замовчуванням сокет, створений викликом socket() є клієнтським активним сокетом, за допомогою якого передбачають з'єднання із сервером (особливості використання таких сокетів розглянемо в наступному розділі). Щоб перетворити такий сокет у серверний прослуховувальний (listening), призначений для приймання запитів на з'єднання від клієнтів, необхідно використати системний виклик listen()

```
int listen(int sockfd, int backlog);
```

де: sockfd — дескриптор сокета, пов'язаний з адресою за допомогою bind();

backlog — задає довжину черги запитів на з'єднання, створеної для цього сокета.

Фактично внаслідок виклику listen() створяться дві черги запитів: перша з них містить запити, для яких процес з'єднання ще не завершений, друга — запити, для яких з'єднання встановлене. Параметр backlog визначає сумарну довжину цих двох черг; для серверів, розрахованих на значне навантаження, рекомендують задавати достатньо велике значення цього аргументу:

```
listen(listenfd, 1024);
```

Внаслідок виклику listen() сервер стає цілковито готовий до приймання запитів на з'єднання.

Прийняття з'єднань

Спроба клієнта встановити з'єднання із сервером після виклику `listen()` має завершуватися успішно. Після створення нове з'єднання потрапляє у чергу встановлених з'єднань. Проте, для сервера воно залишається недоступним. Щоб отримати доступ до з'єднання, на сервері його потрібно прийняти за допомогою системного виклику `accept()`:

```
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

де: `sockfd` — дескриптор прослуховувального сокета;

`cliaddr` — покажчик на структуру, у яку буде занесена адреса клієнта, що запросив з'єднання;

`addrlen` — покажчик на змінну, котра містить розмір структури `cliaddr`.

Головною особливістю цього виклику є повернене значення — дескриптор нового *сокета з'єднання* (connection socket), який створений внаслідок виконання цього виклику і призначений для обміну даними із клієнтом. Прослуховувальний сокет після виконання `accept()` готовий приймати запити на нові з'єднання.

```
struct sockaddr_in their_addr = { 0 };
int listenfd, connfd, sin_size = sizeof(struct sockaddr_in);
// listenfd = socket(...), bind(listenfd, ...), listen(listenfd, ...)
connfd = accept(listenfd, (struct sockaddr *)&their_addr, &sin_size);
// використання connfd для обміну даними із клієнтом
```

Виклик `accept()` приймає з'єднання, що стоїть першим у черзі встановлених з'єднань. Якщо вона порожня, процес переходить у стан очікування, де перебуватиме до появи нового запиту. Саме на цьому системному виклику очікують ітеративні та багатопотокові сервери, що використовують сокети. Такі сервери розглядатимемо далі в цьому розділі.

Задання з'єднань на клієнті

Дотепер ми розглядали виклики, які мають бути використані на сервері. На клієнті ситуація виглядає інакше. Після створення сокета за допомогою `socket()` необхідно встановити для нього з'єднання за допомогою системного виклику `connect()`, після чого можна відразу обмінюватися даними між клієнтом і сервером.

```
int connect(int sockfd, const struct sockaddr *saddr, socklen_t alen);
```

де: `sockfd` — сокет, створений за допомогою `socket()`;

`saddr` — покажчик на структуру, що задає адресу сервера (IP-адресу віддаленого хоста, на якому запущено сервер і порт, який прослуховує сервер);

`alen` — розмір структури `saddr`.

Ось приклад створення з'єднання на клієнті:

```
struct sockaddr_in their_addr = { 0 };
sockfd = socket(...);
// заповнення their_addr.sin_family, their_addr.sin_port
inet_aton("IP-адреса-сервера", &(their_addr.sin_addr));
connect(sockfd, (struct sockaddr *)&their_addr, sizeof(their_addr));
// обмін даними через sockfd
```

У разі виклику connect() починають створювати з'єднання. Повернення відбувається, якщо з'єднання встановлене або сталася помилка (при цьому поверненим значенням буде –1). Зазначимо, що, якщо connect() повернув помилку, користуватися цим сокетом далі не можна, його необхідно закрити. Коли потрібно повторити спробу, створюють новий сокет за допомогою socket().

Клієнтові зазвичай немає потреби викликати bind() перед викликом connect() – ядро системи автоматично виділяє тимчасовий порт для клієнтського сокета.

Закриття з'єднань

Після використання всі з'єднання необхідно закривати. Для цього застосовують стандартний системний виклик close():

```
close(sockfd);
```

Обмін даними між клієнтом і сервером

Після того як з'єднання було встановлене і відповідний сокет став доступний серверу (клієнт викликав connect(), а сервер – accept()), можна обмінюватися даними між клієнтом і сервером. Для цього використовують стандартні системні виклики read() і write(), а також спеціалізовані виклики recv() і send():

```
// прийняти nbytes або менше байтів із sockfd і зберегти їх у buf
ssize_t recv(int sockfd, void *buf, size_t bytes_read, int flags);
// відіслати nbytes або менше байтів із buf через sockfd
ssize_t send(int sockfd, const void *buf, size_t bytes_sent,int flags);
```

Виклики recv() і send() відрізняються від read() і write() параметром flags, що задає додаткові характеристики передавання даних. Тут задаватимемо цей параметр, що дорівнює нулю:

```
int bytes_received = recv(sockfd, buf, sizeof(buf), 0);
int bytes_sent = send(sockfd, buf, sizeof(buf), 0);
```

Важливою особливістю обміну даними між клієнтом і сервером є те, що send() і recv() можуть отримати або передати меншу кількість байтів, ніж було запитано за допомогою параметра nbytes, при цьому така ситуація не є помилкою (особливо часто це трапляється для recv()). Для відсилання або отримання всіх даних у цьому разі необхідно використати відповідний системний виклик повторно:

```
// отримання даних обсягом sizeof(buf)
for (pos = 0; pos < sizeof(buf); pos += net_read)
    net_read = recv(sockfd, &buf[pos], sizeof(buf)-pos, 0);
printf("від сервера: %s", buf);
```

У разі помилки ці виклики повертають –1. Серед кодів помилок важливими є ECONNRESET (віддалений процес завершився негайно, не закривши з'єднання) і EPIPE (для send() це означає, що віддалений процес завершився за допомогою close(), не прочитавши всіх даних із сокета; у цьому випадку також буде отримано сигнал SIGPIPE).

Виклик recv(), крім того, може повернути нуль. Це означає, що з'єднання було коректно закрите на іншій стороні (за допомогою виклику close()). Так у коді сервера можна відстежувати закриття з'єднань клієнтами:

```
net_read = recv(sockfd, ...);
if (net_read == 0) { printf("з'єднання закрите\n"); }
```

Структура найпростішого ітеративного сервера

Розглянемо приклад розробки найпростішого *луна-сервера*, що негайно повертає клієнтові всі отримані від нього дані (в UNIX-системах така служба доступна за стандартним портом із номером 7).

Опишемо основний цикл сервера (інший код є стандартним — підготовка структур даних, виклик `socket()`, `bind()` і `listen()`).

У головному циклі спочатку необхідно прийняти з'єднання. Після цього в циклі зчитують дані із сокета з'єднання і відсилають назад клієнтові. Цей внутрішній цикл триває доти, поки клієнт не закриє з'єднання або не буде повернено помилку. У кінці ітерації головного циклу сокет з'єднання закривають:

```
for(; ;) {
    sockfd = accept(listenfd,
        (struct sockaddr *)&their_addr, &sin_size);
    printf("сервер: з'єднання з адресою %s\n",
        inet_ntoa(their_addr.sin_addr));
    do {
        bytes_read = recv(sockfd, buf, sizeof(buf), 0);
        if (bytes_read > 0) send(sockfd, buf, bytes_read, 0);
    } while (bytes_read > 0);  // поки сокет не закритий
    close(sockfd);
}
close(listenfd);
```

Зазначимо, що в даному прикладі сервер можна перервати тільки за допомогою сигналу (Ctrl+C, `kill()` тощо), при цьому за замовчуванням прослуховувальний сокет закрито не буде. Коректне закриття сокета може бути зроблене в оброблювачі сигналу або у функції завершення.

Недолік такого сервера очевидний — поки обробляються дані, нові клієнти не можуть створити з'єднання (не виконується `accept()`). Далі розглянемо, як можна уникнути цієї проблеми.

Структура найпростішого клієнта

Приклад луна-клієнта, що відсилає серверу дані, введені із клавіатури, і відображає все отримане у відповідь, наведено нижче.

Сокет і виклик `connect()` створюються стандартним способом. Обмін даними відбувається в нескінченному циклі (для виходу потрібно ввести рядок "вихід", після чого з'єднання буде коректно закрите). Зазначимо, що для простоти весь код повного отримання даних наведено тільки для `recv()`:

```
// sockfd = socket(...), connect(sockfd, ...)
for (; ;) {
    fgets(buf, sizeof(buf), stdin);
    if (strcmp(buf, "вихід\n") == 0) break;
    stdin_read = strlen(buf);
    send(sockfd, buf, stdin_read, 0);
    for (pos = 0; pos < stdin_read; pos += net_read)
        net_read = recv(sockfd, &buf[pos], stdin_read - pos, 0);
    printf("від сервера отримано: %s", buf);
}
close(sockfd);
```

Структура найпростішого багатопотокового сервера

Розглянемо, як можна вирішити проблему ітеративного сервера відповідно до підходів, описаних у розділі 15. У цьому розділі зупинимося на розробці найпростішого багатопотокового сервера, а у розділі 16.4.4 — на введенні-виведенні з повідомленням на базі виклику select().

Відмінності для багатопотокової реалізації фактично торкнуться тільки основного циклу сервера. У ньому необхідно приймати з'єднання і створювати для його обробки окремий потік у неприєднаному стані (замість очікування завершення його виконання, потрібно очікувати нових з'єднань). Дескриптор сокета передають як параметр у функцію потоку.

```
for(; ;) {
    connfd = accept(listenfd, ...);
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_create(&tid, &attr, &process_request, (void *) connfd);
}
```

Функція потоку перетворює параметр до цілочислового типу, виконує обмін даними із клієнтом і закриває сокет з'єднання:

```
void *process_request(void *data) {
    int connfd = (int) data;
    // ... обмін даними із клієнтом через connfd
    close(connfd);
}
```

16.4.4. Введення-виведення з повідомленням

Системний виклик select()

У розділі 15.5.3 ішлося про особливості введення-виведення із повідомленням і загальні принципи використання системного виклику select(). Зараз наведемо приклад використання select() для повідомлення про стан сокетів у реалізації вже відомого луна-сервера.

Насамперед зауважимо, що select() в UNIX працює з усіма файловими дескрипторами, доступними для процесу (серед них можуть бути дескриптори сокетів). Ці дескриптори, як зазначалося в розділі 13, — цілі числа із послідовною нумерацією від нуля. Таке відображення визначає низку особливостей використання виклику select().

Опишемо синтаксис select():

```
#include <sys/select.h>
#include <sys/time.h>
int select(int fd_count, fd_set *readset, fd_set *writeset,
     fd_set *exceptset, const struct timeval *timeout);
```

де: fd_count — кількість дескрипторів, зміну стану яких потрібно відстежувати (оскільки дескриптори нумеруються від нуля, це число дорівнює максимальному номеру дескриптора плюс один);

readset, writeset — набори дескрипторів, для яких відповідно потрібно відстежувати готовність до читання і записування даних (роботу із такими наборами буде розглянуто окремо);

timeout — покажчик на структуру, що задає максимальний час очікування під час виконання цього виклику (NULL — нескінченне очікування).

Цей виклик повертає кількість дескрипторів, які змінили свій стан, або значення –1 у разі помилки.

Перед викликом select() необхідно задати набори дескрипторів для відстеження готовності до дій, які становлять інтерес. Якщо не потрібно відстежувати готовність до певної дії (читання або записування), замість відповідного набору, слід передати нульовий покажчик.

Тип fd_set є непрозорим для користувача (хоча прийнято відображати набір дескрипторів у вигляді бітового масиву, де позиції окремих бітів відповідають номерам дескрипторів, і говорити про задання і очищення біта для дескриптора). Доступ до наборів дескрипторів здійснюють за допомогою макросів:

```
// очищення всіх бітів fdset
void FD_ZERO(fd_set *fdset);
// задання біта в наборі fdset для дескриптора fd
void FD_SET(int fd, fd_set *fdset);
// очищення біта в наборі fdset для дескриптора fd
void FD_CLR(int fd, fd_set *fdset);
// перевірка, чи увімкнуто біт для fd у наборі fdset
int FD_ISSET(int fd, fd_set *fdset);
```

Наприклад, для задання набору і увімкнення біта для дескриптора потрібно виконати такий код:

```
fd_set fdarr;
FD_ZERO(&fdarr);
fd = socket(...);   // або open()
FD_SET(fd, &fdarr);
```

Крім того, змінні типу fd_set можна присвоювати одна одній.

Коли біт для дескриптора увімкнуто перед викликом select(), це означає, що застосування збирається відстежувати готовність до відповідної дії для цього дескриптора. Під час виконання select() усі біти для дескрипторів, які не готові до виконання дії, очищаються. У результаті, коли біт виявляється увімкнутим після виклику select(), це означає, що такий дескриптор готовий до виконання дії:

```
// очікування готовності до читання дескрипторів із набору fdarr
select(20, &fdarr, NULL, NULL, NULL);
// тут можна досліджувати fdarr
if (ISSET(fd, &fdarr)) {
    // дескриптор fd готовий до читання
}
```

Розглянемо особливості визначення готовності до дій для дескрипторів на прикладі сокетів. Готовність до читання визначають як наявність даних, які можна прочитати із сокета з'єднання, наявність запиту на нове з'єднання для прослуховувального сокета або закриття з'єднання під час читання (коли recv() повер-

тає нуль). Готовність до записування визначають як можливість записати дані в сокет з'єднання або закриття з'єднання під час записування (коли send() генерує SIGPIPE).

Розробка сервера на основі введення-виведення з повідомленням

Опишемо особливості розробки луна-сервера на основі системного виклику select(). Спочатку потрібно визначити основний набір дескрипторів сокетів та його копію:

```
fd_set fdarr, fdarr_copy;
```

Максимальний номер дескриптора, готовність якого потрібно відстежувати, у Linux задають як результат виконання системного виклику getdtablesize(), що повертає розмір таблиці дескрипторів процесу:

```
int listenfd, connfd, fd;      // дескриптори сокетів
int maxfd = getdtablesize();
```

Тепер потрібно створити прослуховувальний сокет listenfd і викликати для нього функції bind(), connect() і listen() (ці дії є стандартними).

Після виконання всіх дій з listenfd його необхідно помістити в набір дескрипторів. Далі, під час виконання функції select(), поява запитів на з'єднання на цьому сокеті свідчитиме про готовність до читання даних.

```
// listenfd = socket(...), bind(listenfd ...), listen(listenfd ...)
FD_ZERO(&fdarr);
FD_SET(listenfd, &fdarr);
```

Розглянемо, навіщо потрібна копія fdarr. Річ у тому, що набір дескрипторів під час виконання select() змінюється так, що стає непридатним для передавання в цей виклик. Задані біти в наборі тепер відповідають тим дескрипторам, які готові до виконання дій, а не тим, для яких необхідно відстежувати цю готовність. Доцільно організувати код так, щоб увімкнуті біти fdarr завжди відповідали дескрипторам, для яких потрібно відстежувати готовність. Для цього fdarr формують під час роботи сервера, але ніколи не передають у select(). Перед кожним викликом select() створюють копію fdarr, і саме її змінюють усередині select() і мають за основу для перевірки готовності дескрипторів.

Наведемо структуру основного циклу сервера. Його тіло складається із трьох основних частин:

- створення копії набору дескрипторів і виклику select();
- обробки запитів на нові з'єднання (вона полягає у перевірці біта для прослуховувального сокета і прийманні з'єднання, якщо біт увімкнуто);
- обходу набору дескрипторів поточних з'єднань для перевірки готовності до читання і виконання власне читання.

```
for (; ;) {
    fdarr_copy = fdarr;   // створення копії набору дескрипторів
    select(maxfd, &fdarr_copy, NULL, NULL, NULL);
    // обробка запиту на нове з'єднання
    // перевірка сокетів, готових до читання, на підставі fdarr_copy
}
```

Нове з'єднання з'являється, коли прослуховувальний сокет опиняється у стані готовності до читання. У цьому разі необхідно прийняти нове з'єднання (викликати accept()) і додати сокет з'єднання у fdarr (увімкнувши у ньому відповідний біт). Тепер готовність до читання буде перевірено і для цього з'єднання. Після завершення цієї дії необхідно перейти до наступної ітерації циклу (тобто до виклику select()):

```
if (FD_ISSET(listenfd, &fdarr_copy)) {
    connfd = accept(listenfd,
        (struct sockaddr *)&their_addr, &sin_size);
    FD_SET(connfd, &fdarr); // додати дескриптор до основного набору
    continue;               // повернутися до select()
}
```

Перевірку сокетів з'єднання виконують обходом усіх наявних дескрипторів (крім прослуховувального сокета) і перевіркою кожного з них на готовність до читання. Якщо дескриптор сокета до читання готовий, дані із нього читають за допомогою recv() і відсилають назад викликом send(). Якщо recv() повернув нуль, то це означає, що клієнт закрив з'єднання. У такому разі необхідно закрити відповідний дескриптор сокета і вимкнути відповідний біт у наборі. Далі цей сокет не перевірятимуть.

```
for (fd = 0; fd < maxfd; fd++) {
    if (fd != lfd && FD_ISSET(fd, &fdarr_copy)) {
        bytes_read = recv(fd, buf, sizeof(buf), 0);
        if (bytes_read <= 0) {
            close(fd);
            FD_CLR(fd, &fdarr);
        }
        else send(fd, buf, bytes_read, 0);
    }
}
```

Отже, у наборі fdarr підтримують у ввімкнутому стані біти для прослуховувального сокета і всіх активних сокетів з'єднання.

16.4.5. Використання доменних імен

Системні виклики інтерфейсу сокетів використовують IP-адреси. Щоб отримати доступ до віддаленого хоста на основі доменного імені, потрібно це ім'я попередньо розв'язати, відіславши запит на локальний DNS-сервер за допомогою розпізнавача.

Найбільш використовуваними системними викликами інтерфейсу розпізнавача є gethostbyname() і gethostbyaddr(). Перший з них застосовують для перетворення доменних імен в IP-адреси (IPv4), другий — для перетворення IP-адрес у доменні імена. Ці виклики потребують підключення заголовного файла <netdb.h>. Тут обмежимося розглядом функції gethostbyname():

```
struct hostent *gethostbyname(const char *hostname);
```

де hostname — рядок із доменним іменем (коротке або повне ім'я).

Ця функція повертає покажчик на розміщену у пам'яті структуру hostent, що містить інформацію про всі IP-адреси, які відповідають цьому доменному імені. Коли під час виконання запиту до локального DNS-сервера сталася помилка, буде повернуто нульовий покажчик і задано змінну h_errno.

Структура hostent містить такі поля:

- h_name — канонічне ім'я цього хоста (повне ім'я, задане A-записом у базі даних DNS);
- h_aliases — масив імен, які є аліасами цього хоста (їм відповідають CNAME-записи в базі даних DNS);
- h_addr_list — масив покажчиків на IP-адреси, що відповідають цьому імені (це цілочислові значення із мережним порядком байтів, придатні для перетворення в структуру in_addr).

Останніми елементами h_aliases і h_addr_list є нульові покажчики.

Розглянемо приклади. Для початку наведемо код, що відображає канонічне ім'я хоста і список його IP-адрес:

```
struct hostent *he; char **pp;
he = gethostbyname("DNS-ім'я");
if (he != NULL) {
    printf("Повне ім'я: %s\n", he->h_name);
    for (pp = he->h_addr_list; *pp != NULL; pp++) {
        struct in_addr *addr = (struct in_addr *)(*pp);
        printf("IP-адреса: %s\n", inet_ntoa(*addr));
    }
}
```

Покажемо приклад задання адреси сокета на основі доменного імені.

```
struct sockaddr_in their_addr = { 0 };
struct hostent *he = gethostbyname("DNS-ім'я");
their_addr.sin_addr = *((struct in_addr *)he->h_addr_list[0]);
```

Змінна h_errno може, наприклад, набувати значення HOST_NOT_FOUND (хост із даним доменним іменем не знайдено).

16.4.6. Організація протоколів прикладного рівня

Потокові та дейтаграмні сокети — це інтерфейс між застосуванням і протоколом транспортного рівня (TCP або UDP). Звідси очевидно, що реалізацію протоколу прикладного рівня здійснює саме застосування на базі інтерфейсу сокетів. Висвітлимо деякі особливості реалізації таких протоколів [75].

Приклад розробки простого протоколу

Прикладом такого протоколу може бути протокол зв'язку програми-чату, яка використовує потокові сокети. Кожне повідомлення від користувача має складатися із двох частин: імені користувача і тексту повідомлення. Сервер відсилає отримані повідомлення всім іншим користувачам. Вихідний потік даних від сервера може мати такий вигляд:

```
ІванПривітМиколаДо побачення
```

Зауважимо, що повідомлення мають змінну довжину. Завдання полягає в тому, щоб дати можливість клієнтам у загальному потоці байтів, отриманих від сервера, виділяти окремі повідомлення і відокремлювати в них ім'я користувача від тексту повідомлення.

Перший підхід до розв'язання цього завдання полягає в тому, щоб зробити всі повідомлення однієї довжини (із доповненням до неї, наприклад, нульовими символами). Це спричиняє пересилання мережею значного обсягу непотрібної інформації навіть у разі не дуже великих повідомлень.

Коректнішим підходом у даному разі є інкапсуляція даних через розробку *формату пакета* із виділенням у ньому:

- заголовка, що містить довжину пакета;
- поля `name` (ім'я користувача фіксованої довжини);
- поля `chatdata` (текст повідомлення змінної довжини).

Фактично угода про формат пакета визначає мережний протокол.

Сервер формує пакет на підставі повідомлення користувача (таку підготовку до пересилання мережею називають також *маршалізацією* — marshaling), клієнт виділяє повідомлення з пакета (робить *демаршалізацію* — demarshaling).

Приймемо за правило, що довжина поля `name` становить 8 байт (коротші імена користувачів доповнюють до цієї довжини нульовими символами). Далі вважатимемо, що максимальна довжина тексту повідомлення (поля `chatdata`) становить 240 байт. Звідси випливає, що максимальна довжина пакета (248 байт) може бути відображена заголовком завдовжки 1 байт.

Розглянемо, який вигляд матимуть пакети для наведених раніше даних. Перший пакет (повідомлення від Івана):

```
0E                  B2 E2 E0 ED 00 00 00 00   CF F0 E8 E2 B3 F2
(довжина – 14 байт) І в а н (доповнення)  П р и в і т
```

Другий пакет

```
13                  CD E8 EA EE EB E0 00 00   C4 EE 20 EF EE E1 ...
(довжина – 19 байт) М и к о л а           Д о   п о б ...
```

Зазначимо, що загалом довжина може бути подана кількома байтами, у цьому разі слід звертати увагу на те, що для пересилання мережею вони повинні бути перетворені до мережного порядку байтів.

Розглянемо отримання пакетів клієнтом. Клієнтові доступна довжина поточного пакета (із неї починаються дані пакета), крім того, відома максимальна довжина (249 байт, включаючи заголовок). У застосуванні необхідно виділити буфер, достатній для розміщення двох пакетів максимальної довжини:

```
char buf[498];
```

У цьому буфері відбуватиметься реконструкція пакетів у міру їхнього надходження. Після кожного отримання даних за допомогою `recv()` їх поміщають у буфер і перевіряють, чи весь пакет отримано.

```
if (buffer_len > 1 && buffer_len >= (buffer[0] + 1))
    // пакет отримано повністю
```

Тут buffer_len — поточний обсяг даних у буфері, buffer[0] — перший байт буфера, що містить довжину пакета (крім байта заголовка). Перевірку на те, чи отримано більш як один байт даних, роблять тому, що тільки в цьому разі можна визначити довжину пакета. Коли пакет отримано повністю, його можна використовувати у застосуванні, після чого поверх нього у буфер почнуть записувати наступний пакет.

Особливою ситуацією є отримання за recv() блоку даних, що містить інформацію із двох пакетів (останні байти одного і перші байти іншого). У результаті буфер міститиме повний пакет і неповну частину ще одного пакета (саме тому у буфері зарезервоване місце для двох пакетів). Оскільки довжина першого пакета відома, можна визначити, скільки байтів буфера належить другому пакету:

```
second_bytes = buffer_len - (buffer[0] + 1);
```

Далі можна обробити перший пакет, перемістити дані другого пакета на початок буфера і відкоригувати buffer_len:

```
strncpy(buf, buf + buffer_len, second_bytes);
buffer_len = second_bytes;
```

Тепер буфер знову готовий прийняти дані, отримані за допомогою recv().

Особливості реалізації прикладних протоколів на прикладі HTTP

Наведені принципи роботи із протоколами прикладного рівня справедливі й для складніших протоколів, які використовують на практиці. Деякі особливості реальних протоколів розглянемо на прикладі HTTP.

Блок заголовків HTTP-пакета, на відміну від описаного простого протоколу, має змінну довжину. Завершення цього блоку визначають за наявністю у потоці даних комбінації символів "\r\n\r\n". Окремі елементи блоку заголовків (його рядки) розділяються символами "\r\n".

Є два типи HTTP-пакетів зі спільними принципами побудови: запит (request) і відповідь (response). Перший рядок запиту містить команду для сервера (метод), адресу документа на сервері та версію протоколу. У цьому разі метод GET визначає запит на отримання документа із сервера на основі його адреси:

```
GET /test.html HTTP/1.1
```

Перший рядок відповіді містить версію протоколу і код відповіді (наприклад, код 200 означає коректне повернення даних):

```
HTTP/1.1 200 OK
```

Інші рядки блоку заголовка у специфікації HTTP називаються просто заголовками і мають формат ім'я_заголовка: значення_заголовка. Наприклад, версія протоколу HTTP/1.1 задає для запиту обов'язковий заголовок Host:, який визначає доменне ім'я комп'ютера-сервера, задане клієнтом:

```
Host: server.com
```

За блоком заголовків може слідувати тіло пакета. Є два способи визначити його довжину. Перший спосіб універсальний і рекомендований до використання. Для цього у блоці заголовків задають заголовок Content-Length:, значенням якого

є довжина тіла пакета. Програмі-клієнту потрібно виділити довжину із цього заголовка:

```
// header_buf – буфер із даними блока заголовків
clen_str = strstr( header_buf, "Content-Length: ");
if (clen_str != NULL)
    sscanf(clen_str, "Content-Length: %d\r\n",&content_length);
```

Подальші дії аналогічні до розглянутих раніше для простого протоколу.

Другий підхід зводиться до того, що сервер після пересилання HTTP-пакета закриває з'єднання. Для цього в запиті необхідно задати заголовок

```
Connection: close
```

Клієнту потрібно відстежити кінець з'єднання (коли `recv()` поверне нуль). Цей підхід простіший у реалізації, але прийнятний не для всіх випадків, оскільки часто клієнт (браузер) запитує сторінки не по одній, а групами (наприклад, HTML-сторінку і всі графічні файли, на які вона посилається), і пересилання всієї групи сторінок за допомогою одного з'єднання дає змогу заощаджувати ресурси і час.

Тіло пакета може бути відсутнім (наприклад, його немає в запиті методом GET).

Розглянемо приклад формування HTTP-запиту:

```
char request[] = "GET /index.html HTTP/1.1\r\n"
                 "Host:server.com\r\n\r\n";
// ...з'єднання через стандартний порт HTTP (порт 80)
send(sockfd, request, sizeof(request), 0);
// ...обробка відповіді
close(sockfd);
```

16.5. Архітектура мережної підтримки Linux

Тут зупинимося на деяких особливостях реалізації мережної підтримки в Linux, зокрема на базовій мережній архітектурі цієї ОС, особливостях підтримки інтерфейсу сокетів, а також формуванні пакетів під час реалізації пересилання і отримання даних.

16.5.1. Рівні мережної підтримки

Мережна архітектура Linux складається із кількох рівнів (рис. 16.5).

- На верхньому рівні перебувають прикладні програми. Зв'язок цих програм із ядром здійснено через програмний інтерфейс сокетів, описаний у попередньому розділі.
- Інтерфейс сокетів складається із двох рівнів: універсального рівня сокетів Берклі і рівня INET-сокетів, специфічного для підтримки TCP/IP. Про них ітиметься в наступному розділі.
- Нижче від інтерфейсу сокетів розташовані засоби підтримки різних рівнів мережної архітектури:
  - транспортного, які відповідають за підтримку протоколів транспортного рівня (TCP і UDP);

- мережного, які відповідають за підтримку протоколів мережного рівня (насамперед, IP);
- канального, які відповідають за формування пакетів канального рівня (наприклад, Ethernet-фреймів) і за відображення адрес канального рівня на IP-адреси.

- На найнижчому рівні перебувають драйвери мережних пристроїв, що підтримують пристрої конкретного типу (наприклад, Ethernet-адаптери).

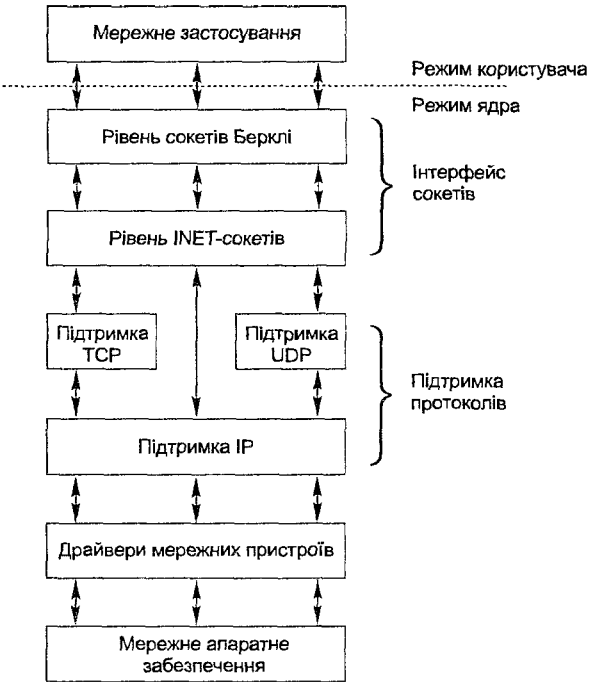

Рис. 16.5. Мережна архітектура Linux

Зазначимо, що хоча мережним пристроям в Linux, як і в більшості UNIX-систем, надають спеціальні імена (наприклад, перший Ethernet-адаптер має ім'я eth0), ці імена не відповідають файлам на файловій системі. Основне завдання драйвера мережного пристрою — пересилання даних із ділянок пам'яті ядра у пам'ять мережного пристрою і назад.

16.5.2. Підтримка інтерфейсу сокетів

Рівні підтримки інтерфейсу сокетів

Ядро Linux розрізняє два рівні підтримки інтерфейсу сокетів.

- На верхньому перебуває *інтерфейс сокетів Берклі*, реалізований за допомогою універсальних *об'єктів сокетів Берклі* (структур типу socket). Об'єкти сокетів Берклі містять інформацію, що не залежить від реалізації конкретної

мережної архітектури. Із такими об'єктами працюють реалізації системних викликів інтерфейсу сокетів (`socket()`, `bind()`, `listen()`, `send()` тощо).

- На нижньому перебувають реалізації інтерфейсу сокетів, що залежать від конкретної мережної архітектури. Прикладом є *інтерфейс INET-сокетів*, що складається з об'єктів INET-сокетів (структур типу `sock`), які містять інформацію, специфічну для стека протоколів TCP/IP. Далі як архітектурно-залежні сокети розглядатимуться тільки INET-сокети.

Об'єкти сокетів

Об'єкти сокетів Берклі реалізовані як файли: для кожного сокета ядро створює індексний дескриптор на спеціальній файловій системі `sockfs`. Крім інформації, що належить до файлової системи, такий об'єкт містить:

- тип сокета (потоковий або дейтаграмний);
- стан сокета (зі з'єднанням або без нього);
- універсальні операції сокетів, що відповідають базовим системним викликам інтерфейсу сокетів Берклі (`bind()`, `listen()`, `connect()`, `accept()` тощо);
- покажчик на відповідний об'єкт INET-сокета.

Об'єкт INET-сокета, в свою чергу, містить:

- локальні та віддалені IP-адреси, номери портів та іншу інформацію, необхідну для підтримки TCP/IP;
- черги пакетів, які одержані із сокета і очікують відсилання через сокет;
- чергу очікування для процесів, що очікують на цьому сокеті (наприклад, внаслідок виконання блокувального системного виклику `recv()`);
- реалізації універсальних операцій сокетів для стека TCP/IP.

Реалізація універсальних операцій об'єкта сокета Берклі звичайно зводиться до виклику відповідних методів пов'язаного з ним об'єкта INET-сокета.

16.5.3. Пересилання і отримання даних

Формування пакетів

Кожний пакет, який ядро Linux передає карті мережного інтерфейсу, має стандартну структуру, показану на рис. 16.2. Цей пакет формують на кількох рівнях мережної архітектури Linux.

Код мережної підтримки Linux зберігає кожний пакет разом із усіма заголовками у спеціальній ділянці пам'яті – *буфері сокета* (socket buffer). Із кожним таким буфером пов'язаний дескриптор (структура `sk_buff`), що містить покажчики на об'єкт INET-сокета, заголовки різних рівнів і корисне навантаження пакета, а також поля зв'язку для об'єднання таких структур у списки (наприклад, список усіх пакетів сокета).

Ядро не копіює дані між рівнями мережної підтримки, замість цього в усі функції роботи з пакетами передають покажчик на дескриптор буфера сокета. Далі розглянемо, як цей буфер використовують на різних рівнях мережної архітектури Linux.

Реалізація пересилання даних

Зробимо загальний огляд кроків, до яких зводиться пересилання даних мережею внаслідок виконання системного виклику `send()` для сокета.

1. Викликають метод `write()` для об'єкта файлової системи, пов'язаного з об'єктом сокета Берклі. Виконання цього методу зводиться до виклику його реалізації для INET-сокета.
2. Розділяють дані на пакети, які будуть інкапсульовані для передавання мережею відповідно до заданого протоколу. Для кожного пакета виділяють пам'ять під буфер сокета і його дескриптор.
3. Засоби транспортного рівня копіюють корисне навантаження (пакет прикладного рівня) із буфера режиму користувача у буфер сокета, після чого до цього пакета додають TCP або UDP-заголовок.
4. Керування передають засобам мережного рівня, які працюють із тим самим буфером, додаючи до пакета транспортного рівня IP-заголовок.
5. Засоби канального рівня додають інформацію, специфічну для цього рівня (наприклад, формують Ethernet-фрейм у буфері сокета навколо пакета мережного рівня), визначають, як використати мережний пристрій для пересилання цього пакета, і поміщають буфер сокета із підготовленим пакетом у чергу пакетів цього пристрою.
6. Драйвер мережного пристрою отримує керування, коли буфер сокета із пакетом поміщають у чергу для цього пристрою. При цьому перевіряють, чи готовий мережний пристрій переслати дані зараз. Коли це неможливо, операцію планують як відкладену і виконують планувальником пізніше, коли можливо — дані копіюють із буфера сокета у пам'ять пристрою (звичайно з використанням DMA).

Реалізація отримання даних

Почнемо з виконання системного виклику `recv()`. Цей виклик спричиняє виконання методу `read()` для об'єкта сокета Берклі, а через нього — виконання такого самого методу для INET-сокета. Цей метод перевіряє чергу отриманих пакетів для цього сокета і, якщо вона порожня, призупиняє поточний процес, поміщаючи його дескриптор у чергу очікування для цього сокета.

Тепер розглянемо послідовність дій, внаслідок яких у черзі отриманих пакетів з'являються пакети. Оскільки ініціатором появи даних у цій черзі є мережний пристрій (процес, що виконує `recv()`, перебуває у призупиненому стані), дії розглядаються у зворотному порядку — від мережного пристрою до системного виклику `recv()`.

1. Пакет, отриманий мережним каналом, зберігають у пам'яті мережного пристрою, який при цьому генерує переривання.
2. Оброблювач переривання у драйвері пристрою виділяє пам'ять під буфер сокета і копіює дані (пакет канального рівня) з пам'яті пристрою в цей буфер.
3. Буфер сокета поміщають в загальну чергу пакетів, отриманих від мережних пристроїв. Відкладена операція ядра час від часу передає перший пакет цієї черги засобам мережного і транспортного рівня для обробки.

4. Засоби мережного рівня визначають, чи адресований пакет цьому хосту (у цьому разі його передають засобам транспортного рівня), чи його потрібно переадресувати іншому хосту.
5. Засоби транспортного рівня знаходять INET-сокет, якому адресований пакет (на підставі значення порту, заданого в заголовку транспортного рівня) і поміщають буфер сокета в чергу пакетів для цього сокета. При цьому поновлюють процес, що очікує на операції `read()` для сокета.
6. Операція `read()` вибирає перший пакет із черги для сокета і копіює із нього корисне навантаження (пакет прикладного рівня) у буфер режиму користувача.

16.6. Архітектура мережної підтримки Windows XP

Компоненти мережної підтримки

Розглянемо основні компоненти мережної підтримки Windows XP (рис. 16.6).

Мережні API надають засоби для доступу до мережі із прикладних програм, незалежні від протоколів. Звичайно такі API частково реалізовані в режимі користувача (компоненти їхньої підтримки в режимі ядра – це драйвери, які називають драйверами мережних API). Серед мережних API найширше використовують інтерфейси поіменованих каналів (описаний у розділі 11.6.3), Windows Sockets (описаний далі в цьому розділі) та віддаленого виклику процедур (RPC) (описаний у розділі 20).

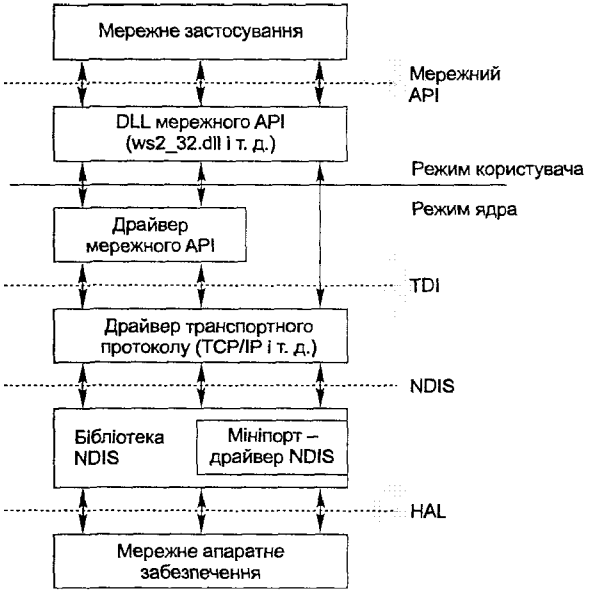

Рис. 16.6. Мережна архітектура Windows XP

Драйвери транспортних протоколів приймають IRP-пакети від драйверів мережних API і обробляють запити, що містяться там, як вимагають реалізовані

в них протоколи (TCP, IP тощо). Під час цієї обробки може знадобитися обмін даними через мережу, додавання або вилучення заголовків пакетів, взаємодія із драйверами мережних апаратних пристроїв. Драйвери протоколів відповідають за підтримку мережної взаємодії організацією повторного збирання пакетів, встановлення їхньої послідовності, відсилання повторних пакетів і підтверджень. Усі драйвери транспортних протоколів надають драйверам мережних API універсальний *інтерфейс транспортного драйвера* (Transport Driver Interface, TDI). Прикладом драйвера транспортного протоколу є драйвер підтримки TCP/IP (tcpip.sys).

Мініпорт-драйвери NDIS відповідають за організацію взаємодії драйверів транспортних протоколів і мережного апаратного забезпечення. Вони взаємодіють з іншими компонентами режиму ядра винятково за допомогою функцій спеціальної бібліотеки NDIS (ndis.sys), що реалізують універсальну *специфікацію інтерфейсу мережного драйвера* (Network Driver Interface Specification, NDIS). Деякі з цих функцій призначені для доступу із драйверів протоколів до засобів мініпорт-драйверів, інші мініпорт-драйвери викликають для безпосереднього доступу до мережних апаратних пристроїв через HAL. Прикладами мініпорт-драйверів NDIS є драйвери конкретних мережних адаптерів.

Спільне використання файлів і принтерів

Основою підтримки спільного використання файлів і принтерів є спеціальний протокол *спільної файлової системи Інтернету* (Common Internet File System, CIFS). Він визначає правила взаємодії з файловими клієнтами і серверами. Реалізація CIFS для Windows XP складається із двох частин – клієнтської (редиректор) і серверної (сервер), реалізованих як драйвери файлових систем.

Драйвер редиректора перехоплює запити на виконання файлового введення-виведення (наприклад, виклик функцій `ReadFile()`, `WriteFile()` або звертання до драйвера принтера), перетворює їх у CIFS-повідомлення і пересилає на віддалений хост, використовуючи драйвер транспортного протоколу. Драйвер сервера приймає повідомлення від драйвера протоколу, перетворює їх назад у запити введення-виведення і передає драйверу локальної файлової системи (наприклад, NTFS) або драйверу принтера. Повернення даних відбувається у зворотному порядку.

Інтерфейс керування драйверами CIFS реалізовано у вигляді фонових процесів `Workstation` (для редиректора) і `Server` (для сервера). Ці процеси перехоплюють запити на створення і зміну *мережних ресурсів* (network shares), які адміністратор системи відкрив для спільного використання (цими ресурсами можуть бути розділи диска, каталоги, принтери тощо). Для керування такими ресурсами можна використовувати утиліту `net.exe`, що входить у поставку Windows XP. Для створення мережного ресурсу її виклик матиме вигляд `net share ім'я_ресурсу=шлях`, де `шлях` задає шлях до локального каталогу, який потрібно зробити розділюваним:

```
c:>net share myshare=c:\mydir
```

Як зазначалося в розділі 11 під час вивчення поіменованих каналів, універсальним способом іменування мережних ресурсів є спеціальні *UNC-імена*: **\\ім'я_машини\ім'я_ресурсу**. У разі використання такого імені активізується спеціальний драйвер (Multiple UNC Provider, MUP), що послідовно відсилає IRP-пакети кожному зареєстрованому у системі редиректору доти, поки один із них не

розпізнає ресурс і не надішле відповіді. Після цього MUP-драйвер кешує інформацію, і далі всі запити, що використовують це UNC-ім'я, негайно відправляються цьому редиректору.

```
HANDLE fh = CreateFile("\\\\myserver\\myshare\\myfile.txt",
    GENERIC_READ, ... );
```

Крім UNC-імен, для доступу до ресурсів можна використати стандартні імена томів або портів принтера. Для керування відображенням мережних ресурсів на такі імена теж використовують утиліту net.exe. Для задання зв'язку ресурсу з іменем тому її виклик матиме вигляд net use ім'я_пристрою ім'я_ресурсу, де ім'я_пристрою задають як символьне позначення тому (C:, D: тощо) або символьне ім'я принтера (наприклад, LPT1:), а ім'я_ресурсу — UNC-ім'я цього мережного ресурсу.

```
c:>net use N: \\myserver\myshare
```

Фактично CIFS можна розглядати як основу реалізації розподіленої файлової системи, оскільки прикладне програмне забезпечення може використати надані нею ресурси як локальні. Виконання виклику ReadFile() для файла N:\myfile.txt у прикладній програмі не залежить від того, чим є N: — розділом локального диска або мережним ресурсом для доступу до спільно використовуваного каталогу віддаленого комп'ютера.

Важливою характеристикою протоколу CIFS є те, що він опублікований як стандарт і може бути реалізований у продуктах сторонніх розробників. Таким продуктом зокрема є Samba, що реалізує підтримку CIFS для UNIX-систем. Samba-клієнти можуть використовувати ресурси, надані серверами лінії Windows XP, Samba-сервери — надавати свої ресурси для використання Windows-клієнтами.

16.7. Програмний інтерфейс Windows Sockets

У цьому розділі йтиметься про мережний API, який найширше використовують із тих, що реалізовані у системах лінії Windows XP. Цей API був побудований на зразок API сокетів Берклі і дістав назву Windows Sockets (скорочено Winsock) [9, 110]. Станом на 2004 рік поточною версією цього API є Winsock 2.2.

16.7.1. Архітектура підтримки Windows Sockets

У режимі користувача Winsock API підтримують кількома системними DLL. Найважливіша з них — ws2_32.dll, де реалізовано основні функції, використовувані у застосуваннях. На рівні ядра підтримку Windows Sockets здійснюють драйвером файлової системи afd.sys, що реалізує операції із сокетами і звертається для пересилання даних до відповідного драйвера транспортного протоколу (звичайно це драйвер підтримки TCP/IP).

Інтерфейс Winsock API посідає таке саме місце у багаторівневій мережній архітектурі, що й інтерфейс сокетів Берклі. Він розташований між рівнем застосувань і транспортним рівнем. У ньому відсутні засоби безпосередньої підтримки протоколів прикладного рівня, вона може бути реалізована на основі інтерфейсу Winsock.

16.7.2. Основні моделі використання Windows Sockets

У програмному інтерфейсі Windows Sockets реалізовані всі основні моделі взаємодії між клієнтом і сервером, розглянуті в розділі 15.5, у тому числі синхронне та асинхронне введення-виведення даних. Реалізація синхронного введення-виведення ґрунтується на сокетах із блокуванням, інтерфейс яких практично не відрізняється від інтерфейсу сокетів Берклі. Їхнє використання можливе, коли не потрібні висока продуктивність і масштабованість застосування. На основі сокетів із блокуванням можна реалізовувати багатопотокові сервери (наприклад, використовуючи технологію «потік для запиту»). Підтримують і реалізацію серверів із повідомленням на основі `select()`, хоча пей підхід також не є ефективним.

Більш ефективною і складнішою технологією, яка також реалізована у Windows Sockets, є *асинхронні сокети*. Є два підходи до їхнього використання. Перший із них використовує повідомлення на основі повідомлень і ґрунтується на функції `WSAAsynchSelect()`. При цьому інформацію про події, що відбулися із сокетом, передають у вигляді повідомлень Windows у віконну процедуру, де вона може бути оброблена (особливості використання повідомлень і розробки віконних процедур описані в розділі 17). За другого підходу – повідомлення на основі подій (що використовує функцію `WSAEventSelect()`) – аналогічну інформацію передають сигналізацією об'єктів подій. Цей підхід з'явився у Winsock 2.

Найефективнішим підходом, реалізованим у Winsock 2, є сокети із перекриттям, що застосовують асинхронне введення-виведення, описане в розділі 15.5.4. Сокети також можна використовувати у поєднанні з портами завершення введення-виведення.

16.7.3. Сокети з блокуванням

Як зазначалося, інтерфейс сокетів із блокуванням майже збігається з інтерфейсом Berkeley Sockets. Проте є деякі відмінності.

- Використання сокетів Берклі вимагає підключення набору різних заголовних файлів. Windows Sockets звичайно потребує лише заголовних файлів `<winsock.h>` або `<winsock2.h>` (який підключають для використання можливостей, реалізованих у версії 2). Під час компонування потрібно задавати бібліотеку підтримки Winsock (**ws2_32.lib** для версії 2).
- Перед використанням будь-яких засобів Windows Sockets необхідно ініціалізувати бібліотеку, після використання – вивільнити зайняті ресурси. Для цього є функції `WSAStartup()` і `WSACleanup()`:

```
#include <winsock.h>
WSADATA wsadata;

WSAStartup(MAKEWORD(1,1), &wsadata);
// ... робота із засобами Winsock
WSACleanup();
```

Перший параметр `WSAStartup()` визначає версію бібліотеки, потрібну застосуванню. Звичайно для його формування використовують макрос `MAKEWORD`, якому передають основний і додатковий номери необхідної версії. Наприклад, аби вказати, що застосування потребує використання Winsock версії 2.0, перший

параметр WSAStartup потрібно задати як MAKEWORD(2,0)). Структуру WSADATA зазвичай ніде, крім виклику WSAStartup(), не використовують.

- Важливою відмінністю Winsock є підхід до розуміння дескрипторів сокетів. Як відомо, інтерфейс сокетів Берклі визначає, що такі дескриптори є звичайними файловими дескрипторами (задані цілими числами, які виділяють процесу послідовно — від менших номерів до більших). Для Winsock це не так: дескриптори сокетів не розглядають як файлові дескриптори, і на їхні значення покладатися не можна. Такі дескриптори належать до спеціального типу SOCKET (а не до типу int, як для сокетів Берклі).
- Оскільки дескриптори сокетів у Winsock відрізняються від файлових дескрипторів, для їхнього закриття не можна використати close(). Замість цього необхідно скористатися спеціальною функцією closesocket().
- У разі помилки виклики Winsock повертають спеціальне значення INVALID_SOCKET і не задають змінної errno (для визначення коду помилки необхідно використати функцію WSAGetLastError()). Обробка помилок у Winsock-застосуваннях має такий вигляд:

```
lsock = socket(...);
if (lsock == INVALID_SOCKET) {
    printf("Помилка із кодом %d\n", WSAGetLastError());
    exit();
}
```

- Windows Sockets не визначає функцію inet_aton(), хоча реалізація inet_ntoa() у цій бібліотеці є. Для переведення крапково-десяткового відображення IP-адреси в цілочислове потрібно скористатися функцією inet_addr():

```
unsigned long addr = inet_addr("IP-адреса");
my_addr.sin_addr = *((struct in_addr *)&addr);
```

Ця функція є також і в інтерфейсі сокетів Берклі, але там її використовувати не рекомендують.

Наведемо базовий код, який використовує Windows Sockets для реалізації найпростішого луна-сервера за принципом «потік для запиту». Очевидно, що відмінності цієї версії від версії для сокетів Берклі є мінімальними.

```
#include <winsock.h>
SOCKET lsock, csock;
WSADATA wsadata;
// ... інші описи, аналогічні до версії для сокетів Берклі

// функція потоку
unsigned int WINAPI process_request (void *data) {
    SOCKET sock = (SOCKET) data;
    // ... обмін даними із клієнтом через sock
    closesocket(sock);
    return 0;
}
// основний код сервера
WSAStartup(MAKEWORD(1,1), &wsadata);
// ... стандартні socket(), bind(), listen()
for(; ;) {  // основний цикл сервера
```

```
    sockfd = accept(...);   // стандартна версія
    HANDLE th = (HANDLE) _beginthreadex (
                NULL, 0, process_request, (void *)sockfd, 0, NULL);
}
WSACleanup();
```

16.7.4. Асинхронні сокети

Розглянемо особливості інтерфейсу асинхронних сокетів, що використовують повідомлення через події. Для кожного сокета, стан якого необхідно відстежувати, необхідно створити об'єкт-подію, що належить до типу WSAEVENT.

```
WSAEVENT event = WSACreateEvent();
```

Після створення події необхідно пов'язати її із сокетом, тобто вказати, які зміни стану цього сокета призводитимуть до сигналізації події. Для встановлення такого зв'язку використовують функцію WSAEventSelect():

```
int WSAEventSelect(SOCKET sock, WSAEVENT event, long net_events);
```

де net_events — бітова маска мережних подій. Кожній події відповідає певний біт цієї маски. Застосування очікуватиме отримання повідомлень про події, біти яких задані в масці. Серед прапорців, що визначають біти для подій, можна виділити:

- FD_READ — готовність до читання даних із сокета;
- FD_WRITE — готовність до записування даних у сокет;
- FD_ACCEPT — наявність нового з'єднання для сокета;
- FD_CLOSE — закриття з'єднання.

```
// сигналізація event у разі спроби читання або записування через sock
WSAEventSelect(sock, event, FD_READ | FD_WRITE);
```

Після виклику WSAEventSelect() застосування може перейти в режим очікування зміни стану об'єктів-подій. Для цього потрібно викликати функцію

```
DWORD WSAWaitForMultipleEvents(DWORD ecount, const WSAEVENT *events,
    BOOL waitall, DWORD timeout, BOOL alertable);
```

де: ecount — кількість подій, зміну стану яких відстежують (як і для WaitForMultipleObjects(), вона не може бути більшою за 64);

events — масив об'єктів-подій;

waitall — якщо TRUE, вихід із функції відбувається у разі сигналізації всіх подій масиву, якщо FALSE — будь-якої з них.

Коли сигналізовано одну подію із масиву, WSAWaitForMultipleEvents() повертає значення, що визначає її індекс у масиві events. Щоб отримати цей індекс, потрібно використати такий код:

```
res = WSAWaitForMultipleEvents(...);
cur_event = res - WSA_WAIT_EVENT_0;
```

Обмеження кількості очікуваних подій вимагає виконання додаткових дій у тому випадку, коли очікують більшу кількість з'єднань. Рекомендовано створювати кілька потоків, у кожному з яких відбувається очікування зміни стану підмножини подій.

Після сигналізації події необхідно визначити, які дії із сокетом її викликали. Для цього використовують функцію

```
int WSAEnumNetworkEvents(SOCKET sock, WSAEVENT event,
    LPWSANETWORKEVENTS net_events);
```

де: sock — сокет, зміна стану якого викликала сигналізацію події;

event — подія, яку було сигналізовано (внаслідок виклику функції стан цієї події буде скинуто);

net_events — покажчик на структуру WSANETWORKEVENTS, яку заповнюють під час виклику функції і яка міститиме інформацію про всі дії із сокетом.

Структура WSANETWORKEVENTS містить такі поля:

- lNetworkEvents — маска мережних подій (значення прапорців аналогічні до розглянутих раніше — FD_READ тощо);
- iErrorCode — масив повідомлень про помилки.

Зазначимо, що в масці мережних подій може бути увімкнуто кілька прапорців. Це означає, що сигналізація була викликана кількома одночасними подіями для сокета.

Після виконання WSAEnumNetworkEvents() можна перейти до обробки запитів введення-виведення на підставі того, які біти увімкнуто в масці мережних подій (наприклад, якщо задано FD_READ, можна перейти до читання даних з відповідного сокета).

Розглянемо, як можна розробити луна-сервер на базі повідомлення про події.

Базова структура такого сервера подібна до розглянутого раніше сервера на основі select(). Необхідно динамічно підтримувати масиви подій і сокетів. Перші елементи цих масивів відповідатимуть прослуховувальному сокету, інші динамічно формуватимуться з появою запитів на з'єднання: після виклику accept() відповідні сокет і подія додаватимуться в масив, у разі розриву з'єднання — вилучатимуться з масиву.

Такі масиви змінюються динамічно, для цього добре підходять структури даних стандартної бібліотеки C++ (такі, як list). У цьому прикладі обмежимося роботою зі звичайними масивами. Нові дескриптори і події додаватимуться в кінець масиву, у разі вилучення з масиву такі елементи зміщуватимуться до його початку.

```
#include <winsock2.h>
const int MAX_CLIENTS = 64;
WSAEVENT events[MAX_CLIENTS];  // довжина масивів обмежена
SOCKET socks[MAX_CLIENTS];
// ...
WSAStartup(MAKEWORD(2,0), &wsadata);   // Winsock 2
// ... lsock = socket(...), bind(lsock, ...), listen(lsock, ...)
// додавання в масиви прослуховувального сокета і події для нього
socks[0] = lsock;
events[0] = WSACreateEvent();
```

Після додавання в масиви інформації про прослуховувальний сокет і відповідну подію необхідно зазначити, що сигналізацію цієї події спричинятиме поява нових з'єднань на сокеті.

```
WSAEventSelect(lsock, events[0], FD_ACCEPT);
```

Очікування сигналізації подій необхідно виконувати в циклі.

```
WSANETWORKEVENTS net_events;
for (; ;) {
    res = WSAWaitForMultipleEvents(
        num_socks, events, FALSE, WSA_INFINITE, TRUE);
    cur_event = res - WSA_WAIT_EVENT_0;
    WSAEnumNetworkEvents(
        socks[cur_event], events[cur_event], &net_events);
    // ... обробка запитів введення-виведення на основі net_events
}
```

Проаналізуємо обробку запитів введення-виведення. Очевидно, що маємо інформацію про сокет, який викликав подію (socks[cur_event]), і про характер цієї події (net_events.lNetworkEvents). У разі появи нового з'єднання на прослуховувальному сокеті інформацію про сокет з'єднання і відповідну подію додають у масиви, після чого сокет пов'язують із цією подією для очікування читання або закриття з'єднання:

```
if (net_events.lNetworkEvents & FD_ACCEPT) {
    csock = accept(socks[cur_event], ...);
    // у кінець масивів додають сокет і нову подію
    socks[num_socks] = csock;
    events[num_socks] = WSACreateEvent();
    // очікування read() або close()
    WSAEventSelect(socks[num_socks], events[num_socks],
            FD_READ | FD_CLOSE);
    num_socks++;
}
```

У разі появи запиту на читання (коли на клієнті були введені дані) виконують дії із сокетом з'єднання, що спричинив цей запит.

```
if (net_events.lNetworkEvents & FD_READ) {
    recv(socks[cur_event], ...);
    send(socks[cur_event], ...);
}
```

Нарешті, у разі закриття з'єднання на клієнті необхідно закрити подію та сокет і вилучити інформацію з відповідних масивів.

```
if (net_events.lNetworkEvents & FD_CLOSE) {
    WSACloseEvent(events[cur_event]);
    closesocket(socks[cur_event]);
    for (i = cur_event+1; i < num_socks; i++) {
        events[i-1] = events[i];
        socks[i-1] = socks[i];
    }
    num_socks-i;
}
```

Перевага цього підходу порівняно із використанням select() полягає насамперед в тому, що програміст має повну інформацію про те, для якого сокета відбулася подія і який її характер, що спрощує програмування і дає змогу досягти більшої ефективності (немає необхідності обходити весь масив дескрипторів).

Висновки

- Під час розгляду сучасних мережних архітектур використовують багаторівневий підхід. Найрозповсюдженішою моделлю мережної архітектури є TCP/IP, у якій виділяють чотири рівні: канальний (фізичний), мережний, транспортний і прикладний. Засоби підтримки перших трьох рівнів звичайно реалізують у ядрі ОС.
- Реалізацією мережної архітектури TCP/IP є набір протоколів Інтернету (стек протоколів TCP/IP), який містить, зокрема, протокол IP на мережному рівні та протокол TCP на транспортному рівні. Цей набір протоколів підтримує більшість сучасних ОС.
- Під час передавання пакета його спочатку опускають униз у стеку протоколів операційної системи хоста-відправника (при цьому відбувається його інкапсуляція в пакети протоколів нижнього рівня — спочатку транспортного, потім мережного, потім канального). Потім пакет канального рівня передають фізичною мережею, а після прибуття на хост-одержувач — піднімають у стеку протоколів ОС цього хоста (при цьому відбувається його послідовне демультиплексування із пакета канального рівня, мережного і транспортного). У результаті застосування-адресат отримує пакет протоколу прикладного рівня.
- Ядро ОС може працювати лише з IP-адресами. Для того щоб можна було задавати символьні імена хостів, використовують розподілену систему імен DNS.
- Основою для реалізації доступу із режиму користувача до засобів підтримки стека протоколу TCP/IP є програмний інтерфейс сокетів Берклі. Він розташований між прикладним і транспортним рівнями мережної архітектури TCP/IP. Його аналогом для Windows-систем є інтерфейс Windows Sockets.

Контрольні запитання та завдання

1. У мережі, що складається з Linux- і Windows-хостів, використовують спільний стек протоколів (з реалізацією всіх його рівнів). Перелічіть всі рівні стека протоколів для хостів обох типів, код реалізації яких потребуватиме корекції, якщо розробники ядра Linux змінять інтерфейс сервісу для транспортного рівня. Що потрібно буде коригувати після аналогічної зміни інтерфейсу протоколу для мережного рівня?
2. Які рівні мережної архітектури повинні бути реалізовані у програмному забезпеченні маршрутизатора?
3. Внаслідок обміну даними між хостами A і B зафіксовано, що хост A відправив N IP-дейтаграм, хост B одержав N IP-дейтаграм, хост B відправив M IP-дейтаграм, хост A одержав $4 \times M$ IP-дейтаграм. Поясніть, як це могло статися.
4. Якщо пакет не дійшов за призначенням упродовж певного проміжку часу, протокол TCP забезпечує його повторний запит. Може статися так, що «загублений» пакет надалі «віднайдеться» (наприклад, у випадку, якщо він був затриманий маршрутизатором або проходив мережею з низькою пропускною здатністю) і прийде за призначенням разом із пакетом, отриманим внаслідок повторного запиту. Які засоби протоколу TCP дають змогу вирішити цю проблему?

5. Якщо реалізація протоколу TCP відправляє перші дані після встановлення з'єднання, номер послідовності для них визначають на підставі показів системного годинника. Чому не можна завжди починати пересилання даних через з'єднання з того самого номера послідовності (наприклад, з нуля)?
6. Кожна IP-дейтаграма з інкапсульованим TCP-сегментом містить такі 5 полів: інформацію про транспортний протокол (protocol), IP-адресу і порт джерела (from_host, from_port), IP-адресу і порт призначення (to_host, to_port). Які системні виклики відповідають за задання і зміну кожного з цих полів у процесі встановлення з'єднання з використанням потокових сокетів?
7. Який із транспортних протоколів (TCP або UDP) краще використовувати як базовий протокол для передачі мультимедіа-даних через Інтернет у реальному режимі (Real Audio і т. д.)?
8. Чи може користувач одночасно працювати з веб-браузером і клієнтом електронної пошти, використовуючи одне модемне з'єднання? Якщо така робота можлива, як розрізняти дані, призначені для кожного з цих застосувань?
9. Як використання кешування може підвищити надійність системи іменування (наприклад, DNS)?
10. Запропонуйте серверні архітектурні вирішення для реалізації таких сервісів:
    а) інтерактивний потоковий, який повідомляє всіх підключених клієнтів про інформацію, одержану від кожного з них. З'єднання можна підтримувати протягом декількох годин, але дані передають упродовж коротких проміжків часу (довжиною кілька мілісекунд). За секунду може бути встановлено до 10 з'єднань. Приклад – інтерактивний чат у локальній мережі;
    б) дейтаграмний (відправлення пакета невеликої довжини у відповідь на кожен запит) з частотою з'єднань до 1–2 за секунду. Приклад – сервіс часу;
    в) дейтаграмний з частотою з'єднань до кількох тисяч за секунду. Приклад – локальна база даних;
    г) потоковий, що приймає дані від різних клієнтів великими фрагментами. Час обробки запиту – до 10 с. Частота з'єднань – до 10 за секунду від різних клієнтів. Приклад – сервер друкування.
11. Розробіть веб-клієнт і веб-сервер з використанням сокетів. Клієнт повинен приймати від користувача (наприклад, у командному рядку) доменне ім'я веб-сервера, номер порту та ім'я запитуваної сторінки, формувати HTTP-запит методом `GET` і відправляти його на сервер. Сервер повинен приймати запит, знаходити відповідний файл сторінки і відправляти його клієнту в складі HTTP-відповіді. Заголовки запиту обробляти не потрібно. Після одержання HTTP-відповіді від сервера клієнт повинен відображати її на екрані. Можливі такі типи сервера:
    а) ітеративний;
    б) який обробляє кожен запит окремим процесом-нащадком;
    в) на основі моделі «потік для запиту» (можна скористатися розв'язком завдання 4 з розділу 15);
    г) який реалізує пул потоків (для Windows XP можна використати порт завершення введення-виведення на основі розв'язку завдання 7 з розділу 15).

Клієнт може використовувати таймер очікування (див. розв'язок завдання 8 з розділу 15).

12. Розробіть клієнт-серверну систему, що реалізує віддалене визначення максимального введеного числа («онлайновий аукціон»). Під час запуску клієнта необхідно вказати доменне ім'я і порт сервера. Далі цей інтерфейс клієнта повинен дозволяти користувачу ввести з клавіатури довільну кількість цілих чисел, супроводжуючи кожне з них іменем (наприклад, "ivanov 100", імена можуть бути різними). Після одержання кожного числа від клієнта сервер повинен обчислювати максимум із чисел, введених дотепер, і, якщо він змінився, сповіщати про це всіх користувачів (за допомогою повідомлення у форматі "новий максимум дорівнює 100, введений користувачем ivanov (адреса *n.n.n.n*, з'єднання 4)"). Введення числа 0 спричиняє розрив з'єднання для цього клієнта (сервер також повинен повідомляти про це користувачів відправленням повідомлення у форматі "клієнт з адресою *n.n.n.n* розірвав з'єднання 1"). Під час розробки коду серверного застосування використовуйте введення-виведення з повідомленням (UNIX) або асинхронні сокети (Windows XP).

Розділ 17

Взаємодія з користувачем в операційних системах

- Засоби термінального введення-виведення
- Командний і графічний інтерфейси користувача
- Процеси без взаємодії з користувачем

У цьому розділі опишемо особливості організації взаємодії із користувачем в операційних системах.

17.1. Термінальне введення-виведення

У цьому розділі зупинимося на базовій технології для організації взаємодії із користувачем в операційних системах – термінальному введенні-виведенні. Хоч історія цієї технології нараховує кілька десятиліть, проте вона продовжує залишатися важливою складовою сучасних ОС.

17.1.1. Організація термінального введення-виведення

Спочатку розглянемо принципи організації термінального введення-виведення, що не залежать від конкретної ОС [44].

Поняття термінала

Історично *термінали* (terminals) використовували для організації багатокористувацької роботи із мейнфреймами або мінікомп'ютерами. Це були апаратні пристрої, що складалися із клавіатури і дисплея, які підключали до комп'ютера через інтерфейс послідовного порту. Особливе поширення такі термінали здобули у 70–80-ті роки: усі розроблені в той час операційні системи включали засоби їхньої підтримки, було створено багато прикладного програмного забезпечення, розрахованого на роботу із ними.

Такі термінали працюють у текстовому режимі, за якого обмін даними і їхнє відображення на програмному рівні відбуваються посимвольно. Для відображення використовують екран розміром у символах (звичайно 25 на 80), причому відображатися можуть тільки стандартні символи із кодами ASCII. За подання на екрані розширеного набору символів із кодами 127–255, зокрема символів кирилиці, відповідає символьна таблиця, яка використовується терміналом.

Є спеціальні символи (керуючі коди) і послідовності символів, які не відображаються, а керують виведенням на екран термінала. До керуючих кодів належать такі символи, як повернення каретки, переведення рядка, Backspace тощо. Керуючі послідовності називають також *ESC*-послідовностями (вони починаються із символу із ASCII-кодом 27 – ESC). Передаючи такі послідовності термінала, можна переміщати курсор у довільну позицію екрана, керувати яскравістю відображення символів, для деяких моделей терміналів – кольорами тощо.

Емуляція термінала

У сучасних умовах апаратні термінали застосовують рідко. Проте інтерфейс зв'язку з терміналами (термінальне введення-виведення) не втратив свого значення й досі. Це пояснюється тим, що текстовий режим роботи дуже зручний для розв'язання багатьох задач (організації адміністрування системи, віддаленого доступу до неї тощо), а також широким вибором програмного забезпечення, яке використовує цей режим.

Для спрощення організації термінального введення-виведення у сучасних ОС широко використовують *емуляцію термінала*. Програмне забезпечення (емулятор) приймає дані згідно із домовленостями щодо обміну із відповідним терміналом і відображає на дисплеї комп'ютера інформацію згідно керуючих послідовностей, визначених для термінала (для цього може бути виділене окреме вікно на графічному екрані). У результаті програмне забезпечення, розраховане на роботу із терміналом, можна використовувати із таким емулятором без змін. Одночасно у системі може бути запущено кілька емуляторів термінала (наприклад, кожен у своєму вікні), і користувач має змогу перемикатися між ними, чергово виконуючи введення-виведення. Найпоширеніша емуляція термінала vt100, програмне забезпечення, розраховане на використання цього термінала, із великою ймовірністю працюватиме із будь-яким емулятором.

Надалі говоритимемо про введення із термінала і виведення на термінал, не уточнюючи, що обмін даними майже завжди відбуватиметься не з апаратним терміналом, а з його емулятором.

Віддалені термінали і консоль

Емулятор термінала може бути запущений на віддаленому комп'ютері, при цьому необхідно забезпечити обмін даними мережею між ним і програмним забезпеченням. Прикладом розв'язання такої задачі є протокол telnet, який працює поверх TCP/IP. Відповідний сервер (telnet-сервер) запускають на машині, яка надає віддалений доступ. Він перехоплює дані, що їх застосування передають на термінал, і пересилає їх на віддалену систему. Там працює емулятор термінала (telnet-клієнт), який інтерпретує отримані дані й, у свою чергу, відсилає серверу інформацію, введену на віддаленій машині. Сервер доставляє її застосуванням.

З іншого боку, якщо дисплей комп'ютера, на якому запущена ОС, працює у текстовому режимі, за відображення інформації на ньому теж може відповідати емулятор термінала. У багатокористувацьких ОС із мережним доступом (наприклад, в UNIX) такий термінал часто називають *консоллю*, на відміну від терміналів, які використовують для доступу до системи через мережу.

Термінальне введення

Є два підходи до організації термінального введення.

- У *режимі без обробки*, або *неканонічному режимі* (non-canonical mode), дані передають програмі без зміни (включаючи керуючі коди, такі, як переведення каретки або Backspace). За інтерпретацію цих кодів відповідає програма. Такий режим складніший у використанні (потрібно інтерпретувати керуючі коди), але більш гнучкий. Найчастіше його використовують текстові редактори.
- У разі використання *режиму з обробкою*, або *канонічного режиму* (canonical mode), дані додатково оброблятимуться перед тим як надійти у програму. Така обробка відбувається після натискання користувачем клавіші Enter (введення символу переведення рядка), при цьому керуючі коди буде інтерпретовано і відповідно до них змінено весь уведений рядок (наприклад, якщо в ньому тричі поспіль трапиться Backspace, ці три символи і ще три, введені перед ними, із рядка будуть вилучені). Такий режим простіший для програміста, у програму в даному разі потрапляє вже підготовлений символьний рядок.

Прикладом програмного забезпечення, що реалізує компроміс між цими режимами, може бути розповсюджена в UNIX-системах бібліотека `readline`. Вона надає розширені засоби редагування введеного рядка, які потребують підтримки неканонічного режиму, але її програмний інтерфейс аналогічний до введення в канонічному режимі (у програму потрапляє підготовлений внаслідок редагування рядок).

Введені із клавіатури символи зберігаються у буфері, навіть у неканонічному режимі (трапляються ситуації, коли застосування не може відразу прийняти дані від клавіатури, і потрібно зберегти їх до того моменту, коли з'явиться така можливість). У канонічному режимі дані із буфера передаються програмі після введення символу переведення рядка, у неканонічному – як тільки програма буде готова їх прийняти. Звичайно для кожного термінала створюється свій окремий буфер введення. У разі заповнення буфера може бути виділена додаткова пам'ять.

У більшості випадків уведені символи відразу відображаються на екрані (робота в режимі луни). Можливе відключення цього режиму, наприклад під час введення паролів або команд текстових редакторів.

Термінальне виведення

Для виведення на термінал теж використовують буферизацію. Буфер виводу заповнюють у тому випадку, коли термінал не готовий прийняти символ; у міру його готовності символи із буфера передають терміналу. Відображаючи дані, він інтерпретує керуючі послідовності, після чого показує інформацію, виділяє кольорами окремі ділянки, переміщає курсор тощо.

Головна проблема полягає в тому, що різні модифікації терміналів сприймають різні набори послідовностей. Для її вирішення у сучасних ОС звичайно створюють базу даних терміналів, що містить список терміналів і послідовності, які відповідають кожному із них. В UNIX-системах таку базу називають `terminfo`.

Логічна структура термінального введення-виведення показана на рис. 17.1.

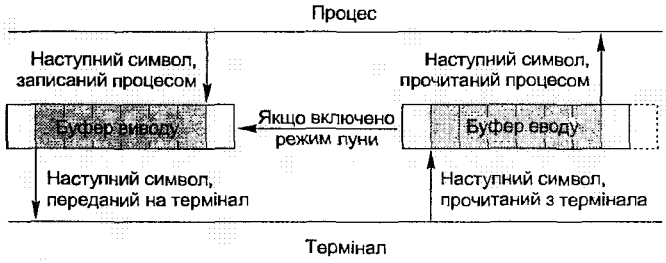

Рис. 17.1. Термінальне введення-виведення

17.1.2. Термінальне введення-виведення в UNIX та Linux

Тут розглянемо особливості реалізації та використання термінального введення-виведення в UNIX-системах на прикладі Linux.

Файли термінальних пристроїв і консоль

Кожному терміналу в Linux (як і в інших UNIX-системах) відповідає файл символьного пристрою. Наприклад, файли /dev/tty*n* (*n* = 1, 2, ..., 63) відповідають терміналам віртуальної консолі (доступний набір таких терміналів, що дає можливість відкривати кілька паралельних сесій користувача; для перемикання між віртуальними консолями використовують комбінації клавіш Ctrl+F*n*), файли /dev/ttyS*n* — терміналам, пов'язаним із з'єднаннями через послідовний порт. Відкривши такий файл, можна працювати із відповідним терміналом.

```
tty2 = open("/dev/tty2", O_RDWR, 0644);
write(tty2, "Виведення на другу віртуальну консоль\n", ...);
```

Консоль Linux емулює спеціальний вид термінала, який називають Linux. Він надає доволі широкі можливості щодо керування відображенням інформації (підтримку кольору, керуючих клавіш, перевизначення символьної таблиці «на ходу»). Поточну консоль відображають файлом /dev/console.

Псевдотермінали

Раніше вже йшлося про принцип роботи протоколу telnet. Виникає запитання: яким чином telnet-сервер перехоплює дані, що їх застосування відсилають на термінал? Для відповіді потрібно ознайомитися із концепцією *псевдотерміналів*.

Псевдотерміналом (pty) називають спеціальний пристрій, який створює і контролює процес режиму користувача (*ведучий процес*, pty master). Для всіх інших процесів (*ведених процесів*, pty slaves) цей пристрій має вигляд реального термінала. У результаті всі дані, якими ведені процеси обмінюються із псевдотерміналом, опиняються під повним контролем ведучого процесу. Зокрема, ведучим процесом у разі telnet є telnet-сервер, веденим — процес, який запускають у telnet-сесії. У результаті сервер має змогу перехоплювати всі дані, які будуть згенеровані під час сесії, та відсилати їх мережею.

Псевдотермінал відображають двома спеціальними файлами пристроїв: *файлом ведучого* (pty master file) і *файлом веденого* (pty slave file). Із файлом ведучого працює ведучий процес, усі інші процеси працюють із файлом веденого. Усі дані,

записані у файл веденого, можуть бути зчитані із файла ведучого і навпаки. У Linux є різні домовленості на імена для цих файлів, наприклад, файли ведучого можуть бути згенеровані за запитом у каталозі /dev/pts.

Є кілька цікавих застосувань псевдотерміналів. Так, є утиліта screen, що перехоплює весь ввід-вивід інтерактивної програми або сесії користувача і зберігає його у файлі. Для цього screen створює псевдотермінал і змушує програму обмінюватися даними не із консоллю, а із цим терміналом.

Керуючий термінал процесу

Процес в UNIX-системі може мати *керуючий термінал* (controlling terminal), з якого отримуватиме сигнали від клавіатури (SIGINT у разі натискання користувачем Ctrl+C, SIGQUIT — Ctrl-D). Звичайно це термінал, із якого ввійшов у систему користувач, що створив такий процес. Для процесу доступний файл /dev/tty, що відповідає цьому терміналу. Далі в розділі ознайомимося із деякими додатковими особливостями взаємодії між процесами і керуючими терміналами.

Наперед визначені файлові дескриптори

Відкривати щоразу файл керуючого термінала під час введення-виведення не дуже зручно. Розглянемо засоби, які надають ОС для спрощення роботи із таким терміналом.

Під час створення нового процесу у його таблиці файлових дескрипторів fd заздалегідь створюють три елементи, котрі використовують як наперед визначені файлові дескриптори. Вони відповідають трьом заздалегідь відкритим файлам, доступним для кожного процесу і за замовчуванням пов'язаних із керуючим терміналом користувача, що створив цей процес:

- stdin — файл стандартного вводу (йому відповідає дескриптор fd[0]);
- stdout — файл стандартного виводу (йому відповідає fd[1]);
- stderr — файл повідомлень про помилки (йому відповідає fd[2]).

Виклик write(1, ...) або write(2, ...) означає виведення на відповідний термінал, read(0, ...) — введення із клавіатури, пов'язаної із цим терміналом. Таку концепцію сьогодні використовують і в інших ОС, які підтримують термінальне введення-виведення. Для можливості перенесення, замість чисел 0, 1 і 2, рекомендують вживати константи STDIN_FILENO, STDOUT_FILENO і STDERR_FILENO:

```
int bytes_read; char buf[1024];
// зчитати дані з файла стандартного вводу
bytes_read = read(STDIN_FILENO, buf, sizeof(buf));
// вивести їх же у файл стандартного виводу
write(STDOUT_FILENO, buf, bytes_read);
```

Є багато прикладних і системних програм, які розраховані на отримання даних з файла стандартного вводу і відображення результатів у файл стандартного виводу. Такі програми називають фільтрами. Серед найвідоміших фільтрів можна виділити sort (сортування файла стандартного вводу, записування результату на стандартний вивід) і grep (пошук заданого підрядка у стандартному вводі, записування рядків, де знайдено цей підрядок, на стандартний вивід).

Програмне керування терміналом

Стандарт POSIX визначає набір системних викликів для керування режимами роботи із терміналом. Для задання атрибутів режиму термінального введення-виведення використовують системний виклик tcsetattr(), а для отримання поточних атрибутів режиму – tcgetattr(). Обидва ці виклики приймають параметром покажчик на структуру termios, яка містить зокрема поле c_lflag – маску прапорців режимів, що керують поведінкою термінала (прапорець ECHO означає роботу в режимі луни):

```
int tcgetattr(int tfd, struct termios *modes);
int tcsetattr(int tfd, int actions, struct termios *modes);
```

де: tfd – дескриптор файла, що відповідає терміналу;

actions – час встановлення режиму (TCSANOW – негайно).

Наведемо приклади використання цих викликів для відключення режиму луни і відновлення попереднього режиму.

```
#include <termios.h>
#include <sys/types.h>
struct termios new_mode, old_mode;
// одержання поточного режиму
tcgetattr(STDIN_FILENO,&old_mode);
new_mode = old_mode;
// відключення режиму луни
new_mode.c_lflag &= (~ECHO);
tcsetattr(STDIN_FILENO,TCSANOW,&new_mode);
// ... введення-виведення з використанням STDIN_FILENO без луни
// відновлення попереднього режиму
tcsetattr(STDIN_FILENO,TCSANOW,&old_mode);
```

17.1.3. Термінальне введення-виведення у Win32 API

Основним для термінального введення-виведення у Win32 є поняття *консолі* [50]. Воно відрізняється від визначеного раніше; фактично консоль – це наданий ОС спеціальний емулятор термінала.

Звичайно консоль пов'язують із конкретним процесом. Для процесів, які запускає ОС, консолі пов'язують із *консольними процесами*, точкою входу для яких є функція main(). Під час виклику CreateProcess() виділення окремої консолі для процесу задається вмиканням прапорця створення CREATE_NEW_CONSOLE. Крім того, кілька процесів можуть спільно використовувати одну й ту саму консоль (наприклад, після виклику CreateProcess() новий процес за замовчуванням успадковує консоль предка).

Логічна структура консолі аналогічна до наведеної на рис. 17.1, за винятком того, що з нею може бути пов'язано кілька буферів виводу. Можна виводити дані у різні буфери, а потім перемикатися між ними.

Для роботи із консоллю є два набори функцій. Функції високого рівня дають змогу працювати зі стандартними вводом та виводом і визначати деякі режими керування консоллю. Функції низького рівня дають можливість застосуванням отримувати повну інформацію про інтерактивну роботу користувача із клавіатурою і мишею. У більшості випадків застосуванню достатньо функцій високого рівня; прикладом застосування, розробленого із використанням функцій низького рівня, є файловий менеджер far.

Наперед визначені дескриптори у Win32

Наперед визначені дескриптори у Win32 пов'язані із консоллю. Відмінність від POSIX полягає в тому, що такі дескриптори не відповідають конкретним цілим числам, а завжди є результатом виклику функції GetStdHandle()

```
HANDLE GetStdHandle(DWORD std_const);
```

Параметр std_const визначає, який дескриптор буде повернуто функцією, і задається як константа STD_INPUT_HANDLE, STD_OUTPUT_HANDLE або STD_ERROR_HANDLE.

```
char buf[1024]; DWORD bytes_read, bytes_written;
HANDLE stdin, stdout;
// одержати стандартні дескриптори
stdin = GetStdHandle(STD_INPUT_HANDLE);
stdout = GetStdHandle(STD_OUTPUT_HANDLE);
// зчитати дані з файлу стандартного вводу
ReadFile(stdin, buf, sizeof(buf), &bytes_read, 0);
// вивести їх же у файл стандартного виводу
WriteFile(stdout, buf, bytes_read, &bytes_written, 0);
```

Програмне керування консоллю

На відміну від UNIX, у Win32 консоллю за замовчуванням володіють лише спеціалізовані консольні застосування або ті, для яких було встановлено прапорець CREATE_NEW_CONSOLE, проте будь-яке застосування, що використовує графічний інтерфейс, може створити собі консоль явно. Для цього використовують функцію AllocConsole(). Після її виклику можна працювати зі стандартними вводом і виводом, вони будуть спрямовані на створену консоль. Після завершення роботи ресурси консолі можна вивільнити, викликавши FreeConsole().

```
AllocConsole();
// ... введення-виведення з використанням дескрипторів,
//     повернутих GetStdHandle()
FreeConsole();
```

Для керування режимами консолі Win32 пропонує аналоги tcgetattr() і tcsetattr() — функції GetConsoleMode() і SetConsoleMode(). Як перший параметр вони приймають дескриптор відкритої консолі, другим є маска прапорців режимів консолі (ENABLE_ECHO_INPUT задає роботу в режимі луни).

```
GetConsoleMode(stdin, &mode);
SetConsoleMode(stdin, mode & ~ENABLE_ECHO_INPUT);
// ... введення з використанням stdin без луни
SetConsoleMode(stdin, mode);   // відновлення режиму
```

17.2. Командний інтерфейс користувача

17.2.1. Принципи роботи командного інтерпретатора

Основним завданням командного інтерпретатора є підтримка інтерактивної роботи користувача, що взаємодіє із системою через термінал.

В UNIX-системах командний інтерпретатор називають *оболонкою* (shell). Розроблено багато версій інтерпретаторів, серед них sh (вихідний варіант), csh (C-shell)

і bash. Інтерпретатор bash входить у стандартну поставку більшості дистрибутивів Linux. Системи лінії Windows XP включають спеціалізований інтерпретатор cmd, який використовують під час роботи в режимі консолі. Роботу інтерпретатора буде розглянуто на прикладі bash [26].

Командний інтерпретатор запускають щоразу, коли користувач реєструється у системі із термінала, при цьому стандартним вхідним і вихідним пристроєм для інтерпретатора і запущених за його допомогою програм є цей термінал. Під час запуску зчитують конфігураційні файли і виконують визначені в них дії із підготовки середовища для цього користувача.

Під час роботи інтерпретатор очікує на введення даних користувача, відображаючи підказку (наприклад, знак долара). Після отримання даних користувача (які формують командний рядок) він інтерпретує їх і виконує деякі дії. Найчастіше вони зводяться до виконання програми, для чого інтерпретатор створює процес, завантажує в нього програмний код і очікує його завершення (відповідно до технології fork+exec). Наведемо приклад.

```
$ cat myfile.txt
вміст файла myfile.txt
$ ... очікування введення
```

Унаслідок виконання цього командного рядка буде створено новий процес, куди буде завантажено код утиліти cat, параметром якої є ім'я файла. Утиліта зчитує цей файл і відображає його на стандартний вивід. Після завершення виконання утиліти інтерпретатор подає підказку і очікує введення наступного командного рядка.

Процес може бути запущений асинхронно, для чого наприкінці командного рядка потрібно задати символ &. Після цього підказка видається негайно, а процес продовжує своє виконання у фоновому режимі.

```
$ updatedb &
$ ... очікування введення, updatedb продовжує виконання
```

Тут утиліта updatedb обновлює базу, що містить імена всіх файлів на файловій системі (для наступного пошуку потрібних імен за допомогою утиліти locate), що є тривалою операцією, яка не потребує втручання користувача. Внаслідок запуску ця утиліта починає виконання у фоновому режимі, а користувач може відразу вводити нові команди.

Набори команд інтерпретатора можуть зберігатися в командних файлах (такі інтерпретатори, як bash, дають можливість використати в них досить потужну мову програмування). Цей командний файл може бути виконаний за тими самими правилами, що і будь-який файл скрипта.

17.2.2. Переспрямування потоків введення-виведення

Важливою технологією, яку реалізують командні інтерпретатори, є переспрямування потоків введення-виведення. При цьому програма замість використання термінала для стандартного потоку введення і стандартного потоку виведення працює із файлом.

```
$ sort > out.txt
$ sort < in.txt
```

Під час виконання таких команд результат виведення опиниться не на екрані, а у файлі out.txt, а введення буде виконано не з клавіатури, а з файла in.txt. При цьому очікування введення із клавіатури не буде.

Можна переспрямовувати одночасно і потік введення, і потік виведення:

```
$ sort < in.txt > out.txt
```

У даному випадку програма зчитує дані з одного файла, обробляє їх і записує в інший файл.

Стандартний дескриптор, що відповідає файлу повідомлень про помилки stderr, при цьому не переспрямовують. Для його переспрямування використовують такий синтаксис:

```
$ sort 2> err.txt
```

Переспрямування потоків введення-виведення у POSIX

Реалізація переспрямування потоків введення-виведення у POSIX ґрунтується на тому, що наперед визначені дескриптори не закріплені за конкретними файлами. Перед створенням процесу або під час його виконання можна змінити відповідний елемент таблиці дескрипторів так, щоб він посилався на інший відкритий файл. У результаті всі системні виклики, що використовують як параметр цей файловий дескриптор, працюватимуть із цим файлом, а не з керуючим терміналом.

Для переспрямування потоків введення-виведення необхідно, щоб наперед визначений файловий дескриптор посилався на той самий файл, що і наявний дескриптор oldfd, відкритий за допомогою open(). Щоб змусити наявний дескриптор посилатися на той самий файл, що й інший відкритий дескриптор, можна використати системний виклик dup2().

```
int dup2(int fd1, int newfd);
```

де: fd1 — вихідний відкритий дескриптор;

newfd — наявний дескриптор, що посилатиметься на той самий файл, що і fd1.

Ось приклад переспрямування потоку виведення:

```
int fd1;
write(STDOUT_FILENO, "на консоль\n", 11);
fd1 = open("output-log.txt", O_CREAT|O_RDWR, 0644);
// переспрямування виведення, аналог myprog > output-log.txt
dup2(fd1, STDOUT_FILENO);
// тут STDOUT_FILENO посилається на той самий файл, що й fd1
write(STDOUT_FILENO, "у файл\n", 7);
close(fd1);
```

Зазначимо, що виклик close() у цьому разі закриє тільки один із двох дескрипторів, STDOUT_FILENO залишиться відкритим, і через нього можна продовжувати роботу із файлом.

Подібний код можна використати і для переспрямування потоків введення-виведення процесу-нащадка (як це робить командний інтерпретатор), тут виклики open() і dup2() виконують у нащадку після виклику fork(), але перед викликом exec().

Переспрямування потоків введення-виведення у Win32

Для реалізації переспрямування потоків введення-виведення в рамках одного процесу у Win32 використовують функцію SetStdHandle()

```
BOOL SetStdHandle(DWORD std_const, HANDLE fh);
```

Тут std_const набуває тих самих значень, що і для GetStdHandle(), а fh відображає відкритий файл.

```
DWORD bytes_written;
HANDLE fd, stdout;
// одержати стандартний дескриптор
stdout = GetStdHandle(STD_OUTPUT_HANDLE);
// вивести дані у файл стандартного виводу
WriteFile(stdout, "на консоль\n", 11, &bytes_written, 0);
fh = CreateFile("output-log.txt", GENERIC_WRITE, ...);
// переспрямування виведення, аналог myprog.exe > output-log.txt
SetStdHandle(STD_OUTPUT_HANDLE, fh);
WriteFile(stdout, "у файл\n", 7, &bytes_written, 0);
```

17.2.3. Використання каналів

Кілька фільтрів можна об'єднувати для обробки даних за допомогою каналів, при цьому стандартний вивід одного процесу переспрямовують на стандартний ввід іншого:

```
$ grep linux * | sort
```

У цьому разі результати виконання утиліти grep передаватимуться на стандартний ввід утиліті sort. Канали можуть об'єднувати будь-яку кількість процесів.

Ще один спосіб використання каналів зводиться до обміну даними між процесом і командним інтерпретатором, під час якого результати виконання процесу будуть підставлені в командний рядок інтерпретатора. Таку технологію називають командною підстановкою (command substitution), для цього командний рядок виклику застосування, результати виконання якого потрібно використати, беруть у зворотні лапки: `виконуваний_файл параметри`.

```
$ grep linux `cat filelist.txt`
```

У даному випадку файл filelist.txt містить список імен файлів, у яких потрібно зробити пошук. Внаслідок виконання утиліти cat список має видаватись на стандартний вивід, але замість цього його підставляють у командний рядок виклику утиліти grep.

Реалізація каналів у POSIX

Для реалізації описаного обміну даними між процесами використовують технологію безіменних каналів (див. розділ 6). Такий канал забезпечує однобічне передавання даних від одного процесу до іншого (тобто коли один процес у канал пише, другий може із нього читати і навпаки).

Безіменний канал відкривають за допомогою системного виклику pipe():

```
#include<unistd.h>
int pipe(int fifo[2]);
```

Тут fifo — масив із двох файлових дескрипторів, що складають канал: один із них використовують для записування, інший — для читання. Один процес може записувати дані у fifo[0], тоді інший зчитуватиме ці дані із fifo[1]; якщо ж перший процес запише у fifo[1], то другий зчитає з fifo[0]. Спроби читання заблo-

кують, якщо в канал не надійшли дані, спроби записування — якщо не було спроби читання.

Одним із способів використання безіменних каналів є переспрямування стандартного потоку введення або виведення нащадка на один із дескрипторів каналу у тому випадку, коли нащадок запускає зовнішнє застосування [24]. При цьому результати виведення застосування можуть бути зчитані у предку із каналу, а дані, які предок записує в канал, стають, у свою чергу, доступні у застосуванні. Розглянемо приклад отримання доступу до результатів виконання іншого застосування (він може бути основою реалізації командної підстановки).

```
int pipefd[2], bytes_read, status;
pipe(pipefd);
if ((pid = fork()) == -1) exit();
if (pid==0) {  // нащадок
   close(pipefd[0]);  // закриття дескриптора для читання
   // переспрямування виведення на дескриптор для записування
   dup2(pipefd[1], STDOUT_FILENO);
   // запуск застосування (execve(...)) або виконання коду нащадка
} else {       // предок
   close(pipefd[1]);  // закриття дескриптора для записування
   // ... читання даних у циклі з pipefd[0]
   close(pipefd[0]);  // закриття дескриптора для читання
   waitpid(pid, &status, 0);
}
```

Безіменні канали Win32

Для створення безіменного каналу у Win32 використовують функцію CreatePipe():

```
BOOL CreatePipe( PHANDLE rpipe, PHANDLE wpipe,
LPSECURITY_ATTRIBUTES psa,  DWORD size );
```

де: rpipe, wpipe — покажчики на дескриптори для читання і записування каналу;

size — розмір буфера для каналу (0 — розмір за замовчуванням).

Розглянемо, як задати переспрямування потоків введення-виведення процесів-нащадків [50].

- Під час створення каналу для нього має бути дозволене успадкування дескрипторів (передаванням у CreatePipe() структури SECURITY_ATTRIBUTES із полем bInheritHandle, рівним TRUE).
- Потрібно задати певні поля структури STARTUPINFO перед її передачею у функцію CreateProcess(). Зокрема, встановлення полів hStdInput, hStdOutput і hStdError спричиняє задання стандартних дескрипторів нащадка. Якщо стандартний дескриптор треба переспрямовувати, йому присвоюють один із дескрипторів каналу, якщо ні — дескриптор, отриманий за допомогою GetStdHandle():

```
// стандартне введення нащадка переспрямовують
si.hStdOutput = wpipe;  // wpipe отриманий з CreatePipe()
// стандартне виведення нащадка не переспрямовують
si.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
```

Крім того, полю dwFlags надається значення STARTF_USESTDHANDLES.

- Під час виклику CreateProcess() параметру bInheritHandles задається значення TRUE. Після виклику необхідно закрити дескриптор для записування (щоб процес-нащадок завершив своє виведення), зчитати дані з дескриптора для читання, після чого його закрити.

```
HANDLE rpipe, wpipe;
STARTUPINFO si = {0}; PROCESS_INFORMATION pi;
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
CreatePipe(&rpipe, &wpipe, &sa, 0);
// переспрямування виведення нащадка на дескриптор для записування
si.hStdOutput = wpipe;
si.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
si.hStdError = GetStdHandle(STD_ERROR_HANDLE);
si.dwFlags = STARTF_USESTDHANDLES;
CreateProcess(NULL, "cmd /c dir", NULL, NULL, TRUE, 0, NULL,
    NULL, &si, &pi);
CloseHandle(pi.hThread);
CloseHandle(wpipe);   // закриття дескриптора для записування
// ... читання даних у циклі з rpipe
CloseHandle(rpipe);   // закриття дескриптора для читання
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
```

17.3. Графічний інтерфейс користувача

Засоби організації графічного інтерфейсу користувача неоднакові для різних ОС. Спільним у них є набір основних елементів реалізації, куди входять вікна з елементами керування (кнопками, смугами прокручування тощо), меню і піктограми, а також використання пристрою для переміщення курсору по екрану та вибору окремих елементів (наприклад, миші).

Розрізняють два основних підходи до реалізації графічного інтерфейсу користувача. Для першого характерна тісна інтеграція у систему засобів його підтримки (вони, наприклад, можуть бути реалізовані в режимі ядра). Другий реалізує підтримку такого інтерфейсу із використанням набору застосувань і бібліотек рівня користувача, який ґрунтується на засобах підсистеми введення-виведення. У цьому розділі як приклад інтегрованої підтримки графічного інтерфейсу користувача буде описано віконну і графічну підсистеми Windows XP, а як приклад реалізації його підтримки в режимі користувача — систему X Window.

17.3.1. Інтерфейс віконної та графічної підсистеми Windows XP

У розділі 2 було розглянуто архітектуру віконної та графічної підсистеми Windows XP. У цьому розділі зупинимося на базових принципах організації програмного коду для цієї підсистеми [44] і наведемо невеликий приклад застосування, яке використовує її можливості.

Базовим елементом Win32-застосування, яке використовує можливості віконної та графічної підсистеми, є вікно, де це застосування може відображати свою інформацію. Кожне вікно належить деякому *класу вікна* (window class), який

реєструється у системі. Програма починає своє виконання із реєстрації класу вікна, після цього об'єкт вікна може бути розміщений у пам'яті та відображений.

Взаємодія із користувачем у застосуваннях, що використовують вікна, відрізняється від тієї, про яку йшлося під час розгляду консольних застосувань. Основним тут є те, що застосування має бути завжди готовим виконати код у відповідь на дії користувача. Наприклад, у будь-який момент може знадобитися перемалювання зображення внаслідок того, що користувач змінив розмір вікна або зробив його активним, вибравши відповідну піктограму мишею на лінійці задач.

Таку постійну готовність складно реалізувати із використанням послідовного виконання коду. У Win32 (і більшості систем, що реалізують графічний інтерфейс користувача) використовують інший підхід до організації програмного коду: в ній реалізовані *застосування, керовані повідомленнями* (message-driven applications). Усі дії користувача (введення із клавіатури, переміщення миші тощо) ОС перехоплює і перетворює у *повідомлення* (messages), які скеровує застосуванню, що володіє вікном, із яким працював користувач. Код застосування містить *цикл обробки повідомлень*, де відбуваються очікування повідомлень та їхні необхідні перетворення, а також оброблювач повідомлень, що його викликають у разі отримання кожного повідомлення. В оброблювачі реалізовано код, який визначає реакцію застосування на ту чи іншу дію користувача. Цикл обробки повідомлень триває доти, поки в нього не потрапить особливе повідомлення, що змушує завершити роботу застосування.

Найпростіший приклад реалізації Win32-застосування із використанням графічного інтерфейсу користувача наведено нижче.

Розробка головної функції застосування

Win32-застосування із підтримкою вікон відрізняються від консольних застосувань, описаних дотепер. Насамперед головну функцію таких застосувань визначають інакше. Її називають `WinMain()`:

```
int WINAPI WinMain( HINSTANCE ih, HINSTANCE ip,
LPSTR cmdline, int cmd_show ) {
    // код головної функції застосування
}
```

де: `ih` — дескриптор екземпляра застосування, який можна використати для ідентифікації виконуваної копії застосування;

`cmdline` — командний рядок, заданий під час запуску застосування;

`cmd_show` — код, що визначає, як передбачено відображати головне вікно застосування (`SW_MAXIMIZE`, `SW_MINIMIZE` тощо).

У коді `WinMain()` насамперед потрібно зареєструвати клас головного вікна застосування. Клас вікна відображають структурою `WNDCLASS`, для якої потрібно задати ряд полів, зокрема:

- `lpszClassName` — ім'я класу, під яким він буде зареєстрований;
- `lpfnWndProc` — адресу процедури вікна, яку система викликатиме з появою повідомлень, адресованих цьому вікну;
- `hIcon` — дескриптор піктограми цього вікна (вона відображатиметься у його заголовку або на панелі задач; для отримання такого дескриптора використовують функцію `LoadIcon()`);

- hCursor — дескриптор курсору, що відображатиметься над вікном; для його отримання використовують функцію LoadCursor());
- hbrBackground — дескриптор спеціального об'єкта (*пензля*, brush), що визначає фоновий колір цього вікна (стандартний колір фону може бути заданий додаванням одиниці до наперед визначеної константи COLOR_WINDOW).

Ось приклад підготовки класу вікна:

```
WNDCLASS wc = { 0 };
wc.lpszClassName = "myclass";
wc.lpfnWndProc = wnd_proc;  // визначення wnd_proc() див. нижче
// стандартна піктограма застосування
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
// стандартний курсор-стрілка
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
```

Після визначення цієї структури покажчик на неї передається у функцію RegisterClass():

```
RegisterClass(&wc)
```

Реєстрація класу вікна дає змогу розміщувати у пам'яті та відображати вікна такого класу. Для розміщення вікна у пам'яті використовують функцію CreateWindow():

```
HWND CreateWindow(LPCTSTR classname, LPCTSTR title, DWORD style,
int x, int y, int width, int height, HWND ph, HMENU mh,
HINSTANCE ih, LPVOID param );
```

де: classname — ім'я, під яким був зареєстрований клас цього вікна;

title — текст, відображуваний у заголовку вікна;

style — стиль вікна, який визначає спосіб його відображення (WS_OVERLAPPEDWINDOW — стандартне вікно із заголовком і керуючим меню);

x і y — координати лівого верхнього кута вікна, width і height — його ширина і висота (ці величини задають у спеціальних віртуальних одиницях, які спрощують масштабування);

ih — дескриптор застосування, що створює вікно (як значення передають відповідний параметр WinMain()).

Ця функція повертає дескриптор вікна (значення типу HWND), за допомогою якого можна дістати доступ до розміщеного у пам'яті об'єкта вікна, наприклад для його відображення.

```
HWND hwnd = CreateWindow("myclass", "Приклад застосування",
WS_OVERLAPPEDWINDOW, 0, 0, 300, 200, NULL, NULL, hinst, NULL );
```

Для відображення вікна використовують функцію ShowWindow():

```
ShowWindow(hwnd, cmd_show);
```

Після того як вікно було відображене, потрібно організувати цикл обробки повідомлень. У ньому необхідно отримати повідомлення за допомогою функції GetMessage(), перетворити його за допомогою функції TranslateMessage() і передати у функцію вікна для подальшої обробки викликом DispatchMessage(). Усі ці функції використовують структуру повідомлення, що належить до типу MSG.

Використання функції TranslateMessage() необхідне під час обробки повідомлень від клавіатури (натискання клавіш перетворюються у повідомлення, що містять введені символи).

Цикл завершується, коли в нього надходить повідомлення завершення (із кодом WM_QUIT), внаслідок чого GetMessage() поверне нуль.

```
MSG msg;
while( GetMessage( &msg, NULL, 0, 0 )) {
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Значенням, яке поверне WinMain(), має бути значення поля wParam структури повідомлення:

```
return (int)msg.wParam;
```

Розробка функції вікна

Функція вікна визначається так:

```
long CALLBACK wnd_proc(HWND hwnd, UINT msgcode, WPARAM wp, LPARAM lp) {
    // обробка повідомлень, адресованих вікну
}
```

де: hwnd — дескриптор відповідного вікна;

msgcode — код повідомлення, що надійшло;

wp, lp — додаткові дані, які можуть супроводжувати повідомлення (відповідають полям wParam і lParam структури повідомлення).

У коді цієї функції перевіряють код повідомлення і залежно від його значення виконують дії.

```
switch (msgcode) {
case код-повідомлення1:
// дії з обробки повідомлення1
case код-повідомлення2:
// дії з обробки повідомлення2 і т. д.
}
```

Є багато різних повідомлень, тут реалізуємо виконання дій у разі одержання двох із них:

- WM_PAINT — надходить вікну щоразу, коли його вміст потрібно перемалювати (у разі відображення, активізації, переміщення тощо);
- WM_CLOSE — надходить у разі закриття вікна користувачем.

Про обробку WM_PAINT ітиметься окремо, а поки що зупинимося на діях після отримання WM_CLOSE та обробці повідомлень за замовчуванням.

У разі отримання повідомлення WM_CLOSE необхідно припинити виконання застосування, для чого потрібно перервати цикл обробки повідомлень. Стандартний спосіб реалізувати таке переривання — скористатися функцією PostQuitMessage(), яка поміщає в цикл повідомлення із кодом WM_QUIT. Як параметр ця функція приймає значення, що стане кодом повернення застосування.

```
switch (msgcode) {
case WM_CLOSE:
```

```
    PostQuitMessage(0);
    return 0;
}
```

Коли у вікно надійшло повідомлення, яке функція вікна не може обробити, у ній потрібно забезпечити обробку такого повідомлення, передбачену ОС (обробку за замовчуванням). Для її реалізації необхідно викликати стандартну функцію вікна, тобто виконати функцію DefWindowProc(), куди передати ті самі параметри, що були отримані функцією вікна.

```
switch (msgcode) {
    // обробка відомих повідомлень
}
// обробка всіх інших повідомлень
return DefWindowProc(hwnd, msgcode, wp, lp);
```

Відображення графічної інформації та контекст пристрою

Залишилося розглянути особливості обробки повідомлення WM_PAINT. Під час цієї обробки потрібно відобразити у вікні певну графічну інформацію. Повідомлення WM_PAINT надходитиме у функцію вікна щоразу, коли таке відображення стане необхідним, наприклад, вікно повідомлення буде виведене поверх інших вікон. Розглянемо, як у Win32 API реалізоване відображення графічної інформації.

Для цього використовують стандартні засоби графічного виведення, які забезпечує GDI. Основною їхньою особливістю є незалежність від пристрою: наприклад, один і той самий код можна використати для відображення інформації на екрані і принтері. Найважливішою концепцією відображення при цьому є *контекст пристрою* (device context).

Такий контекст є внутрішньою структурою даних, що описує поточні властивості пристрою відображення. Перед виконанням відображення застосування має отримати дескриптор контексту пристрою (об'єкт типу HDC) і передати його як один із параметрів у функцію відображення; після виконання цей контекст потрібно вивільнити, внаслідок чого у систему повернуться ресурси, які були потрібні для відображення.

Для отримання контексту пристрою найчастіше використовують дві функції: BeginPaint() викликають із коду оброблювача повідомлення WM_PAINT, GetDC() — в усіх інших випадках. Першим параметром у BeginPaint() передають дескриптор вікна, другим — покажчик на структуру PAINTSTRUCT, яку заповнюють інформацією про ділянку відображення (наприклад, про її розміри). Для вивільнення контексту використовують відповідно функції EndPaint() і ReleaseDC():

```
PAINTSTRUCT ps; HDC hdc;
switch (msgcode) {
case WM_PAINT:
    hdc = BeginPaint( hwnd, &ps );
    // ... відображення графіки з використанням hdc
    EndPaint( hwnd, &ps );
    return 0;
}
```

Після отримання дескриптора контексту пристрою його можна передавати у різні функції відображення графічної інформації (GDI-функції). Їх дуже багато,

для прикладу опишемо використання найпростішої функції відображення тексту — TextOut():

```
BOOL TextOut(HDC hdc, int x, int y, LPCTSTR str, int len);
```

де: hdc — дескриптор контексту пристрою;

x, y — координати точки, із якої почнеться виведення тексту;

str — виведений рядок;

len — довжина рядка без завершального нуля.

Виведення тексту у вікно застосування виглядатиме так:

```
TextOut(hdc, 10, 10, "Hello, World!", 13);
```

17.3.2. Система X Window

У деяких ОС графічні підсистеми не є інтегрованими і не виконуються у режимі ядра. Таким прикладом є *система X Window* (X Window System), яку дотепер широко використовують в UNIX-системах. У цьому розділі наведемо її основні особливості [26, 80].

Базова архітектура системи X Window

Система X Window розроблена із використанням клієнт-серверного підходу, але її базові ідеї відрізняються від традиційних концепцій цієї архітектури.

Насамперед, поняття клієнта і сервера розглядають під незвичним кутом зору. Під клієнтом звикли розуміти застосування, що безпосередньо взаємодіє із користувачем, обробляє команди від клавіатури або миші та відображає дані, а під сервером — застосування, до якого клієнт звертається за даними. У даному випадку це не так (фактично маємо зворотну картину).

В архітектурі X Window під сервером (X-сервером) розуміють програмне забезпечення, що цілковито відповідає за відображення інформації на дисплеї користувача (із використанням графічного адаптера) і за обробку команд від миші та клавіатури. На комп'ютері звичайно запускають лише одну копію X-сервера.

Прикладні програми є клієнтами (X-клієнтами), вони обробляють дані та звертаються до сервера для їхнього відображення. Крім того, сервер перетворює дії користувача (переміщення миші, натискання клавіш на клавіатурі) у дані і передає клієнтам для обробки. Клієнти не мають інформації про особливості роботи із конкретним графічним пристроєм, вони знають лише особливості обміну даними із сервером, зумовлені спеціальним X-протоколом (X Protocol).

Клієнти і сервери можуть перебувати на різних комп'ютерах і виконуватися під різними операційними системами (рис. 17.2). X-протокол звичайно працює поверх TCP/IP, фактично він є набором домовленостей про передавання спеціальних двійкових даних. Один сервер може одночасно відображати інформацію від клієнтів, що перебувають на різних комп'ютерах. Наприклад, комп'ютер під керуванням Linux може відображати засоби редагування відеоінформації, запущені на комп'ютері Silicon Graphics під керуванням ОС IRIX, та адміністративний інтерфейс (вікно додавання нового користувача), запущений на іншій Linux-машині. При цьому жодне із цих застосувань не перебуває у пам'яті комп'ютера, на якому виконується X-сервер. З іншого боку, комп'ютери, на яких запущені X-клієнти,

можуть взагалі не підтримувати графічного відображення інформації — для них досить наявності зв'язку із сервером через мережу.

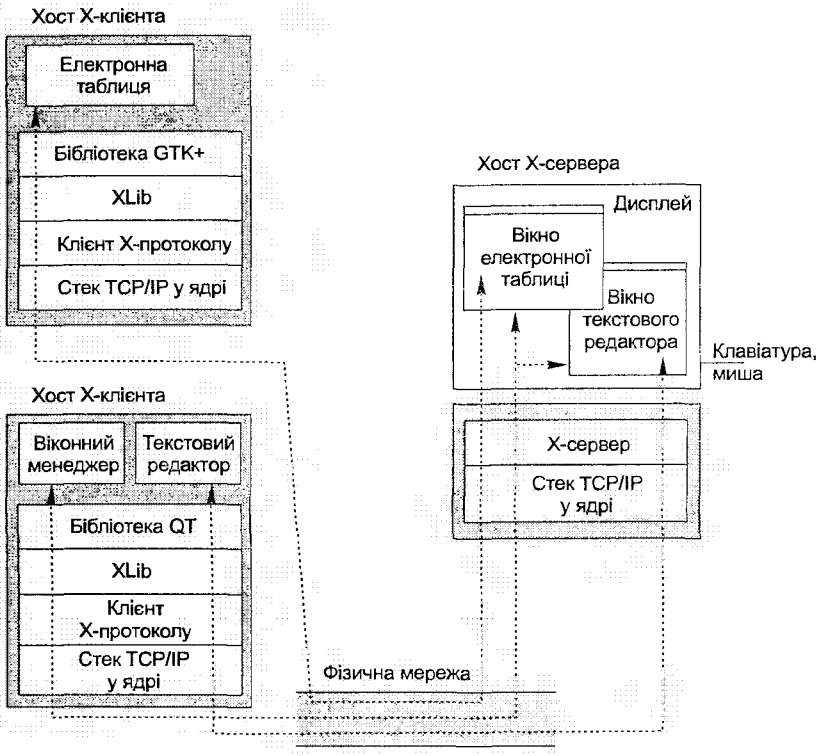

Рис. 17.2. Система X Window

Зазначимо, що використання системи X Window не виключає можливості консольного доступу до ОС. З одного боку, на машині з X-сервером є можливість перемикання між ним і текстовою консоллю, з іншого боку, всі поставки системи X Window включають клієнтську програму `xterm`, що реалізує емуляцію терміналa. Це застосування дає можливість користувачу запускати командний інтерпретатор і консольні застосування на віддаленій машині та відображати результати їхнього виконання на екрані X-сервера.

Віконні менеджери

Команди, надіслані клієнтами сервера, звичайно пов'язані із необхідністю відображення інформації у вікні застосування — ділянці екрана, якою володіє відповідне застосування. Важливою властивістю системи X Window є те, що X-сервер не відповідає за роботу із такими вікнами, для цього використовують спеціальне клієнтське застосування — віконний менеджер (window manager). В обов'язки такого менеджера входить підтримка базових операцій над вікнами (їхнього переміщення, закриття або зміни розмірів), а також «декорування» вікон (відображення рамки, заголовка тощо).

Віконні менеджери майже завжди виконуються на тій самий машині, що й X-сервер, вони запускаються зазвичай відразу після запуску сервера. В усьому іншому це звичайні X-клієнти, які обмінюються із сервером командами, пов'язаними із відображенням вікон, обробкою команд користувача з їхнього переміщення тощо. З іншого боку, вони не відповідають за те, що відбувається всередині вікон застосувань (відображення інформації, обробка дій користувача) — відповідні команди передаються серверу від застосування-клієнта прямо, за їхню генерацію відповідають інструментальні бібліотеки, про що йтиметься далі.

Рішення виділити підтримку віконних операцій в окреме застосування дало змогу досягти додаткової гнучкості роботи із системою. Є багато різних віконних менеджерів, серед найпоширеніших можна виділити fvwm, sawfish, enlightenment. Деякі з них реалізують лише базову функціональність керування вікнами, інші надають користувачу багато можливостей (запуску застосувань, організації робочого столу тощо).

Клієнтські застосування та інструментальні бібліотеки

Теоретично застосування-клієнт може взаємодіяти із сервером із використанням тільки X-протоколу (відкриваючи мережне з'єднання, наприклад, за допомогою сокетів), але на практиці звичайно використовують засоби вищого рівня. Розглянемо їх.

Реалізація системи X Window надає засоби для розробки застосувань-клієнтів. На найнижчому рівні розташована бібліотека XLib, що надає набір функцій, які реалізують базові засоби взаємодії із сервером. Примітиви, надані XLib, дуже складно прямо використати для розробки інтерфейсу користувача. Фактично кожен елемент керування, такий як кнопка або поле вводу, потрібно відображати із використанням найпростіших графічних примітивів (ліній, точок тощо).

На практиці застосування розробляють із використанням набору функцій високого рівня, реалізованих із використанням XLib. Такі функції можуть відповідати за відображення конкретних елементів керування і за обробку дій, виконаних користувачем із цими елементами; вони доступні в рамках *інструментальних бібліотек* (toolkit libraries). Елементи керування в термінології X Window називають *віджетами* (widgets), тому такі бібліотеки називають ще *бібліотеками віджетів* (widget toolkits). Є так само багато інструментальних бібліотек, як і віконних менеджерів, серед найвідоміших можна виокремити Motif (найпоширеніша бібліотека початку 90-х років, не є безкоштовною), Gtk, Qt (дві останні доступні у вихідних кодах, і їх широко використовують у сучасних системах, зокрема в Linux).

Кожна інструментальна бібліотека реалізує свій власний підхід до проектування інтерфейсу користувача: свій набір елементів керування, свої особливості обробки вводу користувача. Застосування, розроблене із використанням такої бібліотеки, відображатиметься на екрані та оброблятиме ввід користувача завжди однаково, незалежно від застосованого віконного менеджера та X-сервера (згадаймо, що все відображення графічного інтерфейсу користувача для застосування зрештою зводиться до генерації команд відповідно до X-протоколу і пересилання їх серверу).

Інтегровані середовища підтримки робочого столу

Як зазначалося, система X Window є дуже гнучкою. Можна використовувати різні віконні менеджери, розробляти застосування із використанням різних інструментальних бібліотек. Ця гнучкість, однак, має свої недоліки.

- Кожний віконний менеджер реалізує керування вікнами по-різному, перехід від одного до іншого зазвичай викликає в користувачів труднощі.
- Ще більше проблем пов'язано із відсутністю стандарту на реалізацію інструментальних бібліотек; у результаті маємо, що в разі переходу від одного клієнтського застосування до іншого доводиться перемикатися між різними підходами до реалізації інтерфейсу користувача, навіть таких його базових компонентів, як прокручування або організація стандартних вікон відкриття файла.
- Кожну інструментальну бібліотеку потрібно поміщати у пам'ять під час завантаження відповідного застосування, що спричиняє додаткові витрати пам'яті у тому випадку, коли одночасно у пам'ять завантажено кілька застосувань, що використовують різні бібліотеки.
- Відсутній стандарт на базові застосування для керування системою, такі як панель керування або файловий менеджер. Є багато реалізацій цих засобів, несумісних за інтерфейсом один з одним.

Для вирішення цих проблем запропоновано концепцію *інтегрованого середовища підтримки робочого столу* (desktop environment). Таке середовище – це інтегрований набір застосувань та інструментальних бібліотек, що реалізують весь набір засобів для організації роботи користувача під управлінням системи X Window. Найчастіше туди входять інструментальна бібліотека, віконний менеджер і базові застосування для керування системою. Фактично, воно реалізує набір компонентів, стандартних для інтерфейсу користувача Windows-систем.

В сучасних ОС широко використовують два інтегровані середовища підтримки робочого столу: KDE і GNOME. Особливості реалізації KDE наведено нижче.

KDE містить:

- інструментальну бібліотеку (Qt), яку використовують для реалізації всіх KDE-застосувань і рекомендують як стандарт для розробників;
- віконний менеджер (kwm), що підтримує керування вікнами на основі домовленостей, прийнятих у Windows-системах;
- набір додаткових бібліотек високого рівня (kdelibs), що реалізують складніші елементи керування (наприклад, діалогові вікна), а також базові системні функції (наприклад, друкування документів);
- набір домовленостей із розробки інтерфейсу користувача;
- набір прикладних програм для розв'язання базових задач повсякденної роботи в системі (веб-браузер і файловий менеджер Konqueror, панель керування control panel, засіб запуску застосувань kpanel, емулятор термінала kconsole тощо).

GNOME має подібну функціональність, важливою відмінністю є те, що це середовище дає змогу вибирати між різними віконними менеджерами, не віддаючи переваги якомусь одному.

17.4. Процеси без взаємодії із користувачем

Дотепер ми розглядали особливості організації взаємодії із користувачем під час розробки інтерактивних процесів. Ще одним видом такої організації є відсутність взаємодії, що становить основну характеристику фонових процесів. Особливості їхньої розробки є темою цього розділу.

17.4.1. Фонові процеси на основі POSIX

У цьому розділі ознайомимося із особливостями розробки фонових процесів у UNIX-сумісних системах [33, 52]. У них фонові процеси за традицією називають *демонами* (daemons). До стандартних демонів Linux належать: `init`, що є предком усіх процесів системи; `cron`, що забезпечує запуск програм у певні моменти часу; `sendmail`, що забезпечує відсилання та отримання електронної пошти тощо.

Сесії та групи процесів

Кожен процес належить до групи процесів, яку позначають ідентифікатором групи процесів (`pgid`). Ядро може виконувати певні дії над усіма процесами групи (наприклад, відсилати їм усім сигнал). Кожна група може мати лідера групи (у цього процесу ідентифікатор (`pid`) збігається з ідентифікатором групи). Звичайно процес успадковує ідентифікатор групи від свого предка, він може також явно змінити свою групу системним викликом `setpgrp()`.

Усі процеси групи володіють одним і тим самим керуючим терміналом у конкретний момент часу, під час виконання він може змінюватися. У групи такого термінала може і зовсім не бути – у цьому разі її процеси не отримуватимуть сигналів від клавіатури.

Кілька груп процесів об'єднують у *сесію*. Для сесії задають ідентифікатор сесії (`sid`). Звичайно процеси сесії відповідають набору процесів, які створив користувач під час інтерактивного сеансу роботи з системою.

Кожний керуючий термінал пов'язаний із сесією та з однією групою процесів цієї сесії (таку групу називають ще *інтерактивною* (foreground) групою, усі інші називають *фоновими* (background), з ними термінали не пов'язані). Протилежне, як і для груп, не завжди вірне – сесія може бути взагалі не пов'язана із керуючим терміналом, у цьому випадку в неї відсутня інтерактивна група.

У кожної сесії є лідер сесії, який відповідає за керування сесією та ізоляцію її від інших; його ідентифікатор збігається з ідентифікатором сесії. Якщо лідер сесії пов'язаний із терміналом, у разі виходу користувача з системи цей процес отримує сигнал `SIGHUP`. Звичайною реакцією на цей сигнал буде завершення роботи процесу-лідера та всіх процесів його сесії.

Коли лідер сесії із терміналом не пов'язаний, сигналу про вихід він не отримає і зможе продовжити виконання після закінчення сеансу роботи. Це особливо важливо для фонових процесів.

Взаємозв'язок між сесіями, групами процесів і керуючими терміналами показана на рис. 17.3.

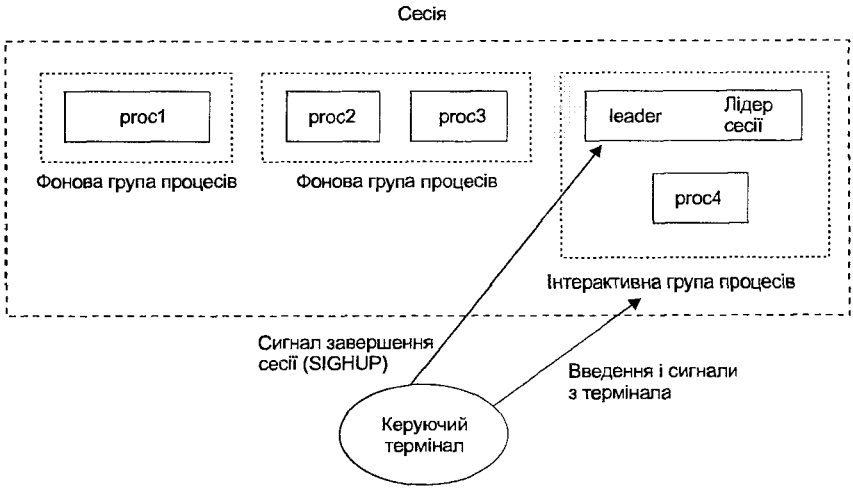

Рис. 17.3. Сесії, групи процесів і керуючі термінали

Створення нової сесії

Подивимось, як користувач може створити нову сесію і для чого це може знадобитися. Нову сесію створюють за допомогою системного виклику `setsid()`. Під час його виконання

- створюють сесію, і поточний процес стає її лідером;
- створюють нову групу процесів у рамках сесії, і поточний процес стає її лідером;
- для поточного процесу розривають зв'язок із керуючим терміналом, якщо він був.

У результаті буде створено сесію без керуючого термінала та з однією фоновою групою всередині неї, що містить на цей момент один процес. Цим як поточний процес, так і його нащадки будуть захищені від сигналів з клавіатури та сигналу про вихід із системи, тобто будуть коректно переведені у фоновий режим. Саме так необхідно реалізовувати фонові процеси, які передбачають використовувати незалежно від наявності інтерактивних сеансів користувачів у системі.

Зазначимо, що, якщо процес уже є лідером групи процесів, виклик `setsid()` призведе до помилки. Для того щоб її уникнути, рекомендують спочатку створити нащадка процесу за допомогою `fork()` (він гарантовано не буде лідером групи), у ньому викликати `setsid()`, а предка завершити.

Розробка фонового процесу

До фонового процесу ставлять такі вимоги: він повинен утворювати власну сесію і групу процесів, не може належати до жодної із сесій і груп користувача та мати керуючий термінал.

Причини висунення цих вимог наведено нижче.

- Демон не може мати керуючого термінала, оскільки не повинен реагувати на переривання у разі спроб введення-виведення із використанням такого термінала.
- Демон має бути лідером фонової групи процесів і лідером нової сесії, щоб він не міг отримувати сигнали (наприклад, у разі натискання Ctrl+C або виходу із системи).

Після запуску демон закриває всі відкриті файли, особливо стандартні потоки введення-виведення, оскільки вони мають бути закриті після виходу користувача із системи, а демон має продовжувати роботу і після цього.

Для того щоб дотриматися всіх вимог, у демоні потрібно виконати певну послідовність дій.

1. Відразу після запуску процес демона має створити нащадка:

```
if ((pid = fork()) < 0) return -1;
```

Це потрібно для того, щоб відразу повернутися в командний процесор, вийшовши з предка на кроці 2. Щоб новий процес гарантовано не міг стати лідером групи процесів, бо він успадковує цю групу від предка – це потрібно пізніше для виклику `setsid()` на кроці 3.

2. Після створення нащадка предок має завершити свою роботу:

```
if (pid != 0) {  // предок
    printf ("демон стартував з pid=%d\n", pid);
    exit (0);
}
```

3. Інші кроки відбуваються в нащадку. У ньому потрібно виконати певну послідовність дій.

- Створити нову сесію:

```
setsid();
```

Поточний процес внаслідок виклику `setsid()`, як було сказано раніше, стає лідером нової сесії, лідером групи процесів і не має керуючого термінала. Головний сенс цього виклику – відключитися від керуючого термінала і втратити зв'язок із поточною сесією, щоб не одержувати ніяких сигналів.

- Змінити поточний каталог на кореневий каталог системи або конкретний робочий каталог демона:

```
chdir("/");
```

Якщо цього не зробити, поточний каталог демона завжди буде тим, з якого він запущений. Тут є ризик, що поточним може виявитися каталог, у який замонтовано файлову систему (у цьому випадку її не можна буде розмонтувати, поки демон не закінчить роботу).

- Можливо, закрити всі відкриті файли (файлові дескриптори):

```
// закрити наперед визначені дескриптори
for (fd = 0; fd < 3; fd++) close(fd);
```

- Перейти в режим очікування (вже було розглянуто різні способи задання очікування у серверних процесах, найпростіший спосіб – виконати виклик `pause()` у циклі):

```
for (; ;) pause();
```

Після запуску демон буде лідером сесії (`pid=sid`), лідером групи процесів (`pid=pgid`) і не матиме керуючого термінала, а його предком стане `init`, оскільки безпосередній предок припинив виконання.

17.4.2. Служби Windows XP

Аналогом демонів у Windows XP є *служби* (services) — фонові процеси, які можуть виконуватися навіть тоді, коли із системою не працює жоден користувач [32, 50]. Із розділу 2 вже відомо про те, що за керування службами відповідає менеджер служб (Service Control Manager). Він приймає керуючі команди від застосувань і відповідно до них виконує дії зі службами (наприклад, запускає на виконання або зупиняє).

Користувацький інтерфейс менеджера служб реалізовано двома способами:

- за допомогою вікна керування службами (Services), яке викликають через підменю Administrative Tools головного меню системи (це вікно відображає список служб, дає змогу запускати і зупиняти окремі служби, дізнаватися про їхні властивості тощо);
- за допомогою утиліти net.exe, що входить у поставку Windows XP; наприклад, команда net start ім'я_служби дає команду менеджерові запустити відповідну службу, net stop ім'я_служби — зупинити її.

Для керування службами необхідно мати адміністративні права у системі.

Програмне встановлення служби

Перед тим як служба стане доступна для менеджера служб, її потрібно встановити у системі. Встановлення зводиться до занесення відповідної інформації у системний реєстр, його можна робити в окремому застосуванні і у коді самої служби, наприклад під час виклику її виконуваного файла із певним параметром командного рядка. Для встановлення служби використовують функцію CreateService():

```
SC_HANDLE CreateService(SC_HANDLE mh, LPCTSTR svc_name,
  LPCTSTR disp_name, DWORD access, DWORD svc_type, DWORD start_type,
  DWORD err_ctl, LPCTSTR exe_path, LPCTSTR lgrp, LPDWORD tag_id,
  LPCTSTR deps, LPCTSTR account, LPCTSTR password);
```

де: mh — дескриптор менеджера служб, отриманий за допомогою виклику функції OpenSCManager();

svc_name — ім'я служби, використовуване для її ідентифікації (зокрема, його передають у функції керування службами, воно є параметром утиліти net.exe тощо);

disp_name — ім'я, відображуване у списку служб вікна керування службами, а також у повідомленнях утиліти net.exe;

svc_type — тип служби (для звичайних служб, виконуваних в окремому процесі, використовують значення SERVICE_WIN32_OWN_PROCESS);

start_type — тип запуску служби (SERVICE_DEMAND_START — запуск на вимогу, SERVICE_AUTO_START — автоматичний запуск під час завантаження системи);

exe_path — повний шлях до виконуваного файла служби (якщо службу встановлюють із її власного коду, для отримання такого шляху можна використати функцію GetModuleFileName());

account — ім'я користувача, із правами якого виконуватиметься служба (якщо NULL — службу виконують із правами спеціального користувача LocalSystem, так роблять найчастіше), password — пароль цього користувача.

Функція повертає дескриптор служби. Після завершення роботи зі службою цей дескриптор потрібно закрити за допомогою функції CloseServiceHandle().

```
char exe_path[1024];
GetModuleFileName(NULL, exe_path, sizeof(exe_path));
SC_HANDLE mh = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
SC_HANDLE sh = CreateService(mh,"mysvc","Служба MySvc",
  SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS, SERVICE_DEMAND_START,
  SERVICE_ERROR_NORMAL, exe_path, NULL, NULL, NULL, NULL, NULL);
CloseServiceHandle(sh);
```

Реалізація коду служби

Перейдемо до безпосередньої реалізації коду служби. Насамперед необхідно визначити кілька змінних. Серед них структура SERVICE_STATUS, що відображає стан служби, дескриптор стану служби (змінна типу SERVICE_STATUS_HANDLE) і прапорець, який визначає, чи можна продовжувати виконання основної функції служби (змінна running). Ці змінні визначаються як глобальні, тому що до них потрібний доступ із кількох функцій (головної функції служби та обробловача команд керування).

```
SERVICE_STATUS status = { 0 };
SERVICE_STATUS_HANDLE sth;
bool running = true;
```

У коді функції main() необхідно визначити масив структур SERVICE_TABLE_ENTRY. Елементами цієї структури є ім'я служби і покажчик на її головну функцію. Для останнього елемента масиву обидва ці поля задаються як NULL. Цю структуру потрібно передати як параметр у функцію StartServiceCtrlDispatcher(). Після цього виконуватиметься код головної функції служби.

```
void main() {
    SERVICE_TABLE_ENTRY disp_table[] =
                {{"mysvc",svc_main},{NULL,NULL}};
    StartServiceCtrlDispatcher(disp_table);
}
```

Головна функція служби

Головну функцію служби викликають під час запуску служби (внаслідок виконання StartServiceCtrlDispatcher()) і виконують до її зупинки. Її код реалізує основну функціональність служби (очікує з'єднань від клієнтів, виконує запити тощо). Визначення цієї функції має такий вигляд:

```
void WINAPI svc_main(DWORD argc, LPTSTR argv[]) {
    // код основної функції служби
}
```

У цій функції потрібно виконати ряд кроків. Насамперед задаються деякі параметри служби встановленням полів глобальної структури SERVICE_STATUS. До них належать:

- dwServiceType — тип служби (SERVICE_WIN32_OWN_PROCESS для звичайних служб, які виконуються в окремому процесі);
- dwCurrentState — поточний стан служби (у цей момент вона очікує початку виконання, і цей параметр покладають рівним SERVICE_START_PENDING);

- dwControlsAccepted – допустимі керуючі команди, які надходитимуть у функцію-обробник команд (SERVICE_ACCEPT_STOP означає, що оброблятиметься тільки команда Stop).

Заповнення структури SERVICE_STATUS виконується так:

```
status.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
status.dwCurrentState = SERVICE_START_PENDING;
status.dwControlsAccepted = SERVICE_ACCEPT_STOP;
```

Після цього необхідно зареєструвати у системі оброблювач керуючих команд цієї служби за допомогою функції RegisterServiceCtrlHandler(), що приймає ім'я служби та адресу функції-обробника, а повертає глобальний дескриптор статусу служби.

```
sth = RegisterServiceCtrlHandler("mysvc",svc_ctrlhandler);
```

Після виконання цього виклику служба матиме змогу реагувати на керуючі команди (наприклад, на команду зупинки служби).

На цьому підготовчий етап завершують. Тепер можна ініціалізувати службу (створити мережні з'єднання, підготувати внутрішні структури даних тощо). Після цього службу переводять у стан виконання, для чого задають нове значення поля dwCurrentState для структури SERVICE_STATUS і викликають функцію SetServiceStatus(), першим параметром якої є глобальний дескриптор статусу служби, а другим – покажчик на структуру SERVICE_STATUS.

```
// ... ініціалізація служби
status.dwCurrentState = SERVICE_RUNNING;
SetServiceStatus (sth, &status);
```

Після цього служба може виконувати будь-які дії, яких від неї вимагатиме програміст (приймати з'єднання від клієнтів тощо).

```
while (running) {
    // ... робота служби
}
```

У даному разі змінна running асинхронно змінюватиметься в оброблювачі команд керування.

Оброблювач команд керування

Функцію-оброблювач команд керування викликають асинхронно у разі отримання службою такої команди (наприклад, за допомогою вікна керування службами або утиліти net.exe). Ця функція схожа на оброблювач сигналів у UNIX або оброблювач консольних команд Windows XP. Вона має приймати один параметр, що відображає команду керування, і виконувати дії залежно від значення цієї команди. У наведеному прикладі ця функція обробляє тільки команду Stop Service (SERVICE_CONTROL_STOP); можна обробляти й інші команди, наприклад Pause/Continue або Shutdown, яку подають у разі припинення роботи ОС. У коді оброблювача змінюють статус служби і значення глобальної змінної running, яке перевіряють в основній функції служби.

```
void WINAPI svc_ctrlhandler(DWORD ctl) {
    if (ctl == SERVICE_CONTROL_STOP) {
        status.dwCurrentState = SERVICE_STOPPED;
```

```
            SetServiceStatus (sth,&status);
            running = false;
        }
    }
```

Вилучення служби

Для вилучення служби із реєстру використовують функцію DeleteService(), куди необхідно передати дескриптор служби. Його потрібно отримати за допомогою функції OpenService(), що як параметри приймає дескриптор менеджера служб та ім'я наявної служби:

```
SC_HANDLE mh = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
SC_HANDLE sh = OpenService(mh,"mysvc",SERVICE_ALL_ACCESS);
DeleteService(sh);
CloseServiceHandle(sh);
```

Зазначимо, що якщо служба в момент вилучення перебувала у стані виконання, її лише позначають для вилучення. Фактичне вилучення відбувається під час наступного завантаження системи.

Висновки

- Термінальне введення-виведення реалізує взаємодію з алфавітно-цифровими пристроями. У сучасних ОС такі пристрої найчастіше є емуляторами терміналa, роботу із якими здійснюють за одними й тими самими правилами. На основі термінального введення-виведення реалізують командний інтерфейс користувача ОС у вигляді командних інтерпретаторів.
- Є різні підходи до організації графічного інтерфейсу користувача, найпоширенішим із них є реалізація такого інтерфейсу як інтегрованої частини системи, що працює в режимі ядра (так зроблено у системах лінії Windows XP), і реалізація засобів його підтримки в режимі користувача у вигляді набору бібліотек та утиліт (прикладом є система X Window).
- Розробка фонових застосувань, що не взаємодіють із користувачем, здійснюється за особливими правилами. В UNIX-системах для таких застосувань закрито можливість інтерактивного обміну даними із користувачем, у системах лінії Windows XP є спеціальний компонент, відповідальний за керування ними.

Контрольні запитання та завдання

1. Наведіть приклад програмного каналу, один із елементів якого має завершитися аварійно через одержання сигналу SIGPIPE. Використовуйте синтаксис каналів командного інтерпретатора.
2. Розробіть застосування для Linux і Windows XP, яке:
   а) створює нащадка, переадресовує свій стандартний вивід на стандартний ввід цього нащадка, виводить повідомлення на стандартний вивід і відновлює попередній стан виводу. Нащадок повинен відображати на стандартний вивід усі дані, отримані на його стандартний ввід;

б) переадресовує стандартний вивід і стандартний потік повідомлень про помилки в кінець файла, ім'я якого задано в командному рядку, після чого виводить повідомлення на стандартний вивід і в стандартний потік повідомлень про помилки;

в) створює двох нащадків і переадресовує стандартний вивід одного з них на стандартний ввід іншого (зв'язуючи їх безіменним каналом). Перший нащадок повинен виводити довільне повідомлення на стандартний вивід, другий – відображати на стандартний вивід усі дані, отримані на стандартний ввід;

г) зберігає стандартний вивід нащадка в рядку символів і відображає вміст цього рядка (так може бути реалізована командна підстановка). Нащадок повинен виводити довільне повідомлення на стандартний вивід.

3. Модифікуйте застосування завдання 2, *б* з розділу 17 так, щоб стандартний потік повідомлень про помилки був спрямований на сервер (заданий IP-адресою і портом) з використанням сокетів. Розробіть сервер, що буде зберігати отриману інформацію у файлі.
4. Модифікуйте командний інтерпретатор для Linux і Windows XP, розроблений під час виконання завдання 10 з розділу 3 і завдання 10 з розділу 14, доповнивши його функцією переадресування введення-виведення, каналів і командної підстановки. Під час розв'язання задачі користуйтеся результатами виконання завдання 2 з розділу 17.
5. Сесія командного інтерпретатора запущена для віддаленого телнет-клієнта. Опишіть, які компоненти ОС на локальному і віддаленому хостах відповідають за відображення стандартного виводу застосувань, запущених у цій сесії.
6. Перелічіть спільні риси та відмінності віконної підсистеми Windows XP і системи X Window. Які переваги і недоліки має кожна із систем?
7. Сесія командного інтерпретатора, запущена на віддаленому хості, відображається у вікні емулятора термінала `xterm`. Опишіть, які компоненти ОС на локальному і віддаленому хостах відповідають за відображення стандартного виводу застосувань, запущених у цій сесії.
8. У якій ситуації стандартний вивід X-клієнта автоматично відображається на текстовій консолі, з якої був запущений X-сервер, а в якій – ні? Чи можна забезпечити відображення виводу X-клієнта на задану текстову віртуальну консоль?
9. Розробіть фонове застосування для Linux і Windows XP, що відстежує всі зміни файлів у заданому каталозі (створення, вилучення, зміну розміру тощо). Ім'я каталогу може бути задане в командному рядку, для Windows XP допустиме його задання в системному реєстрі. Кожну зміну реєструють у файлі у форматі `"час: ім'я_файлу характер_зміни"`.
10. Модифікуйте сервер, одержаний для завдання 11 з розділу 16, реалізувавши його як фоновий процес для Linux і Windows XP.

Розділ 18

Захист інформації в операційних системах

- Організація входу користувачів у систему
- Керування доступом користувачів до даних у UNIX і Windows XP
- Аудит подій у системі
- Безпека даних на локальному комп'ютері та у мережі
- Засоби захисту від атак на систему

У цьому розділі йтиметься про особливості розв'язання головних завдань забезпечення безпеки комп'ютерних систем.

18.1. Основні завдання забезпечення безпеки

Забезпечення безпеки комп'ютерних систем вимагає виконання комплексу завдань, найважливішими серед яких є виконання процедур аутентифікації, авторизації та аудиту, дотримання конфіденційності, доступності та цілісності даних [51].

Аутентифікація

Аутентифікація (authentication) [34] – це процес, за допомогою якого одна сторона (система) засвідчує, що інша сторона (користувач) є тим, за кого себе видає. Під час аутентифікації потрібне свідчення (credentials), що найчастіше складається з інформації, відомої обом сторонам (наприклад, ним може бути пароль). Користувач, що пред'явив коректне свідчення, дістає позитивну відповідь на вимогу аутентифікації.

Авторизація

Після того як аутентифікація відбулась успішно, користувач починає працювати із системою. Тепер йому може знадобитися доступ до різних ресурсів. *Авторизація* (authorization), або *керування доступом* (access control) – це процес, за допомогою якого перевіряють, чи має право користувач після успішної аутентифікації отримати доступ до запитаних ним ресурсів.

Авторизацію здійснюють порівнянням інформації про користувача з інформацією про права доступу, пов'язаною із ресурсом. Зазвичай вона містить тип дії та відомості про користувачів, яким дозволено цю дію виконувати. Якщо користувач має право на запитану дію із цим ресурсом, йому надають можливість її виконати.

Аудит

Під *аудитом* (auditing) розуміють збирання інформації про різні події у системі, важливі для її безпеки, і збереження цієї інформації у формі, придатній для подальшого аналізу. Подіями можуть бути успішні та безуспішні спроби аутентифікації у системі, спроби отримати доступ до об'єктів тощо. Інформацію звичайно зберігають у спеціальному системному журналі (system log). У деяких системах цей журнал має вигляд текстового файла, інші підтримують його спеціальний формат.

Конфіденційність, цілісність і доступність даних

Основним компонентом політики безпеки комп'ютерних систем є дотримання найважливіших характеристик даних.

Конфіденційність (data confidentiality) – можливість приховання даних від стороннього доступу. Її зазвичай забезпечують криптографічним захистом даних за допомогою їхнього шифрування.

Цілісність (data integrity) – спроможність захистити дані від вилучення або зміни (випадкової чи навмисної). Технології підтримки цілісності даних також пов'язані із криптографічними методами, вони включають цифрові підписи і коди аутентифікації повідомлень.

Доступність (data availability) – гарантія того, що легітимний користувач після аутентифікації зможе отримати доступ до запитаного ресурсу, якщо він має на це право. Порушення доступності даних називають *відмовою від обслуговування* (denial of service), однієї із цілей політики безпеки є запобігання випадковому або навмисному доведенню системи до такої відмови.

Про ці завдання докладніше йтиметься далі.

18.2. Базові поняття криптографії

Припустимо, що одна особа збирається відіслати іншій особі повідомлення і хоче бути впевненою в тому, що ніхто інший не зможе його прочитати навіть тоді, коли воно буде перехоплене.

У криптографії повідомлення називають *вихідним текстом* (plaintext), зміну вмісту повідомлення так, що воно стає недоступним для сторонніх, – *шифруванням* (encryption), зашифроване повідомлення – *шифрованим текстом* (ciphertext), а процес отримання вихідного тексту із шифрованого – *дешифруванням* (decryption).

18.2.1. Поняття криптографічного алгоритму і протоколу

Криптографічні алгоритми

Криптографічний алгоритм, який ще називають *шифром* (cipher), – це математична функція, яку використовують для шифрування і дешифрування.

Сучасні криптографічні алгоритми проблему безпеки вирішують виключно за допомогою *ключа* (key). Ключ – це значення, яке використовують для шифрування і дешифрування, при цьому без знання ключа дешифрування має бути технічно неможливим (тобто неможливим за наявності будь-яких доступних ресурсів). Ключ має належати до множини зі значною кількістю елементів: її розмір повинен бути таким, щоб забезпечити технічну неможливість підбору ключа повним перебором усіх елементів цієї множини.

Сукупність алгоритму і множини всіх можливих вихідних текстів, шифрованих текстів і ключів називають *криптосистемою*.

Криптографічні алгоритми, засновані на ключі, поділяють на дві основні групи: *симетричні* (або *алгоритми із секретним ключем*) і *алгоритми із відкритим ключем*. Далі докладніше про особливості цих двох груп.

Криптографічні протоколи

Криптографічний протокол – це послідовність кроків, розроблена для виконання деякої задачі із забезпечення криптографічного захисту даних.

Такий протокол обов'язково включає дві або більше сторін і розглядається як обмін діями між сторонами. Для позначення сторін прийнято використовувати такі імена: *Аліса* – учасник, що робить перший крок; *Боб* – учасник, що робить крок у відповідь.

На окремих етапах протоколів використовують різні криптографічні алгоритми.

18.2.2. Криптосистеми з секретним ключем

Криптографічні алгоритми з секретним ключем

Алгоритми із секретним ключем або симетричні алгоритми – це такі алгоритми, для яких ключ для шифрування повідомлення збігається із ключем для його дешифрування. Справедлива формула

$$D_K(E_K(P)) = P,$$

де P – вихідний текст, E_K – шифрування із ключем K, D_K – дешифрування із ключем K.

Наведемо деякі приклади симетричних алгоритмів.

Алгоритм *DES* (Data Encryption Standard) був прийнятий як національний стандарт США 1977 року. Головним його недоліком є довжина ключа (56 біт), що робить DES недостатньо надійним у сучасних умовах (ключ може бути знайдений за скінчений час перебором усіх можливих ключів). Як альтернативу сьогодні використовують *потрійний DES* (послідовне шифрування трьома різними ключами).

У 2001 році було прийнято новий стандарт на симетричний криптографічний алгоритм. Він (як і сам алгоритм) дістав назву *AES* (Advanced Encryption Standard). За довжиною ключ AES значно перевищує DES (стандарт визначає ключі на 128, 192 і 256 біт), крім того, він відрізняється високою продуктивністю.

Обмін повідомленнями із використанням криптографії із секретним ключем

Подамо у вигляді протоколу обмін повідомленнями у разі використання криптографічного алгоритму із секретним ключем.

1. Аліса і Боб погоджуються, що вони використовуватимуть систему із секретним ключем.
2. Вони доходять згоди щодо спільного ключа.
3. Аліса шифрує вихідний текст ключем і посилає його Бобові.
4. Боб розшифровує повідомлення тим самим ключем.

Другий крок потрібно робити секретно через деякий альтернативний канал передавання даних. Якщо Аліса і Боб не можуть використати такий канал, вони

змушені передавати ключ незашифрованим, інакше інша сторона не зможе ним скористатися. Якщо при цьому ключ перехопить зловмисник, то він далі зможе читати всю інформацію, яку передають каналом.

Цю проблему неможливо вирішити, залишаючись у межах традиційних симетричних алгоритмів. Потрібен принципово інший підхід. Він отримав назву *криптографії із відкритим ключем* (public key cryptography).

18.2.3. Криптосистеми із відкритим ключем

Криптографічні алгоритми з відкритим ключем

Алгоритми з відкритим ключем розроблено таким чином, що ключ, використаний для шифрування, відрізняється від ключа для дешифрування. Ці два ключі працюють у парі: текст, який шифрують одним ключем, дешифрують іншим відповідно до формул

$$D_{Kpri}(E_{Kpub}(P)) = P;$$

$$D_{Kpub}(E_{Kpri}(P)) = P,$$

де E_{Kpub}, D_{Kpub} – шифрування і дешифрування першим ключем пари, E_{Kpri}, D_{Kpri} – шифрування і дешифрування другим ключем пари. Безпека грунтується на тому, що один ключ не може бути отриманий з іншого (принаймні за прийнятний для зловмисника проміжок часу).

Ці алгоритми так називаються тому, що один із цих ключів може бути відкритий для всіх (наприклад, опублікований у пресі), даючи змогу будь-якій особі шифрувати цим ключем, але таке повідомлення може прочитати тільки справжній адресат (що володіє другим ключем).

Ключ, що відкривають для інших осіб для виконання шифрування, називають відкритим ключем (public key), парний до нього ключ для дешифрування – закритим ключем (private key).

Алгоритми із секретним ключем працюють швидше за алгоритми з відкритим ключем, тому, якщо не потрібні специфічні властивості, забезпечувані відкритим ключем (наприклад, не передбачене пересилання даних відкритим каналом зв'язку), достатньо обмежитися алгоритмом із секретним ключем.

Найвідомішим алгоритмом із відкритим ключем є *RSA*.

Обмін повідомленнями з використанням криптографії із відкритим ключем

Обмін повідомленнями у разі використання криптографії з відкритим ключем наведемо у вигляді протоколу.

1. Аліса і Боб погоджуються використовувати систему з відкритим ключем.
2. Боб надсилає Алісі свій відкритий ключ.
3. Аліса шифрує повідомлення відкритим ключем Боба і відсилає його Бобові.
4. Боб розшифровує це повідомлення своїм закритим ключем.

Ця послідовність кроків позбавлена недоліків, властивих для процедури обміну секретним ключем. Перехоплення відкритого ключа зловмисником не спричиняє порушення безпеки каналу, а закритий ключ між сторонами не передають.

18.2.4. Гібридні криптосистеми

На практиці алгоритми із відкритим ключем не можуть цілковито замінити алгоритми із секретним ключем, насамперед через те, що вони працюють значно повільніше, і шифрувати ними великі обсяги даних неефективно.

Для того щоб об'єднати переваги двох категорій алгоритмів, використовують гібридні криптосистеми, де алгоритми із відкритим ключем використовують для шифрування не самих повідомлень, а ключів симетричних алгоритмів, якими далі (після обміну) шифрують весь канал. Такі ключі називають сесійними ключами, бо їх звичайно створюють за допомогою генераторів випадкових чисел для конкретної сесії.

Протокол роботи гібридної криптосистеми наведемо нижче.

1. Боб відсилає Алісі свій відкритий ключ.
2. Аліса генерує випадковий сесійний ключ, шифрує його відкритим ключем Боба і відсилає Бобові.
3. Боб розшифровує сесійний ключ своїм закритим ключем.
4. Обидві сторони далі обмінюються повідомленнями, зашифрованими сесійним ключем.

Гібридні криптосистеми забезпечують якісний захист ключів і прийнятну продуктивність. Про різні застосування цієї технології йтиметься далі.

18.2.5. Цифрові підписи

Ще одна галузь застосування криптографії із відкритим ключем — *цифрові підписи*. Протокол створення і перевірки цифрового підпису при цьому складається з таких кроків.

1. Аліса шифрує документ своїм закритим ключем, тим самим підписуючи його.
2. Аліса відсилає шифрований документ Бобові.
3. Боб розшифровує документ відкритим ключем Аліси, підтверджуючи цифровий підпис.

Цей протокол забезпечує підтримку основних характеристик цифрових підписів, до яких належить, зокрема, неможливість приховати зміну підписаного документа. У цьому разі спроба змінити документ без використання закритого ключа призводить до того, що документ не дешифрується після перевірки підпису.

Алгоритми із відкритим ключем, однак, неефективні для підписування повідомлень великого обсягу. Щоб підвищити продуктивність, цифрові підписи реалізують із використанням односторонніх хеш-функцій.

Односторонні хеш-функції

Хеш-функція — функція, що приймає на вхід рядок змінної довжини, який називають *вхідним образом*, а повертає рядок фіксованої (звичайно меншої) довжини — *хеш*.

За значенням *односторонньої функції* важко віднайти аргумент. *Одностороння хеш-функція* також працює в одному напрямку: легко отримати хеш із вихідного рядка, але технічно неможливо знайти вхідний образ, із якого походить цей хеш. Односторонній хеш-функції властива свобода від колізій: технічно неможливо створити два вхідних образи з одним і тим самим значенням хеша.

Найрозповсюдженішими односторонніми хеш-функціями є MD5 і SHA-1. Крім цифрових підписів, їх можна використати для розв'язання різних задач, які потребують такого відображення важливої інформації, що заслуговує на довіру.

Підписи із відкритим ключем і односторонніми хеш-функціями

У разі використання односторонніх хеш-функцій для цифрових підписів замість документа підписують його хеш (значно менший за обсягом). Протокол набуває такого вигляду.

1. Аліса отримує односторонній хеш документа.
2. Аліса шифрує хеш своїм закритим ключем, тим самим підписуючи документ.
3. Аліса відсилає Бобові документ і зашифрований хеш.
4. Боб розшифровує хеш, переданий Алісою, її відкритим ключем.
5. Боб отримує односторонній хеш переданого Алісою документа.
6. Боб порівнює хеші, отримані під час виконання кроків 4 і 5. Якщо вони збігаються, підпис Аліси можна вважати вірним.

Цей протокол ґрунтується на тому, що підпис хеша можна прирівняти до підпису документа, що випливає із властивості свободи від колізій односторонніх хеш-функцій (неможливо створити два документи, які були б перетворені на один хеш).

18.2.6. Сертифікати

Дотепер ми розглядали обмін повідомленнями між двома сторонами. На практиці значно частіше трапляється ситуація, коли ціла низка сторін домовляються про використання криптографії із відкритим ключем для обміну повідомленнями. У цьому разі доцільно розміщувати відкриті ключі кожної зі сторін у спеціальній базі даних. Протокол відсилання повідомлення набуває такого вигляду.

1. Аліса знаходить відкритий ключ Боба у базі даних.
2. Аліса шифрує повідомлення відкритим ключем Боба і відсилає його Бобові.

Основна проблема при цьому пов'язана із базою даних, у якій зберігають відкриті ключі. Вона повинна мати такі властивості.

- Можливість читати з цієї бази даних може мати будь-який користувач.
- У неї не може записувати дані жоден користувач, за винятком деякої довіреної сторони. У протилежному випадку зловмисник зможе записати у базу свій відкритий ключ поверх ключа легітимного користувача і читати всі повідомлення, адресовані йому.

Водночас, навіть якщо база даних і має такі властивості, зловмисник під час передавання даних може підмінити відкритий ключ Боба своїм ключем. Для того щоб цього уникнути, довірена сторона може поставити цифровий підпис на кожен відкритий ключ.

У реальних застосуваннях довірена сторона (центр сертифікації, Certification Authority, CA) підписує не відкритий ключ, а *сертифікат* — документ, що складається із відкритого ключа та інформації, яка ідентифікує його власника. Після підписання сертифікати заносяться у базу даних. Тепер, коли Аліса отримає із ба-

зи даних сертифікат Боба, вона зможе переконатися у справжності його відкритого ключа верифікацією підпису CA.

На формат сертифікатів є стандарти, наприклад X.509.

18.3. Принципи аутентифікації і керування доступом

Цей розділ присвячено особливостям реалізації аутентифікації і контролю доступу в сучасних операційних системах.

18.3.1. Основи аутентифікації

Аутентифікація надає можливість розрізняти легітимні та нелегітимні спроби доступу до системи. Надійна аутентифікація дає змогу у багатьох випадках обмежити коло потенційних порушників легітимними користувачами системи, спрощуючи цим процедури забезпечення її безпеки.

Свідчення, які вимагаються від користувачів під час аутентифікації, найчастіше зводяться до знання секретної інформації, спільної для користувача і системи (наприклад, пароля). Саме про таку аутентифікацію і йтиметься в цьому розділі. До альтернативних свідчень належать:

- володіння деяким фізичним предметом (наприклад, смарт-картою);
- біометричні параметри (відбитки пальців тощо).

Розрізняють локальну і мережну аутентифікацію. У разі успішної локальної аутентифікації користувач доводить свою легітимність для використання ресурсів однієї комп'ютерної системи (свідчення користувача перевіряють локально), мережна аутентифікація дає змогу користувачу довести легітимність для використання всіх ресурсів мережі (свідчення користувача передають для перевірки на спеціальний сервер із будь-якого комп'ютера мережі).

Облікові записи

Для того щоб аутентифікація користувача була можлива, у системі має зберігатись інформація про цього користувача. Таку інформацію називають *обліковим записом* (account). Із ним звичайно пов'язують такі дані:

- ім'я користувача, яке він вказує для входу у систему;
- ідентифікатор користувача, що зазвичай є чисельним значенням, унікальним у межах комп'ютера або групи комп'ютерів (цей ідентифікатор ОС використовує під час аутентифікації і авторизації);
- інформація про пароль користувача;
- інформація про обмеження на вхід користувача у систему (термін легітимності облікового запису, періодичність зміни пароля, години і дні тижня, у які користувач може отримувати доступ у систему тощо);
- інформація про групи, до яких належить цей користувач;

- місце знаходження домашнього каталогу користувача (у якому він може створювати свої файли);
- налаштування сесії користувача (шлях до його командного інтерпретатора тощо).

Інформацію про облікові записи зберігають у *базі даних облікових записів* (account database). Адміністратор системи може змінювати будь-яку інформацію в цій базі, для інших користувачів звичайно доступна лише зміна їхнього власного пароля.

Групи користувачів

У сучасних ОС для зручності адміністрування системи користувачі можуть об'єднуватись у групи. Користувач може одночасно належати до кількох груп. Під час авторизації доступу до об'єктів перевіряють не тільки права самого користувача, але й права груп, до яких він належить.

Інформацію про групи також зберігають у базі даних облікових записів. Звичайно ОС визначає кілька стандартних груп, які створюють під час її установки, зокрема, групу адміністраторів системи (які можуть виконувати в ній будь-які дії) і групу звичайних користувачів із обмеженим доступом.

Аутентифікація з використанням односторонніх функцій

Для перевірки пароля немає потреби знати цей пароль, досить уміти відрізняти правильний пароль від неправильного. Тому замість зберігання паролів доцільно зберігати односторонні функції цих паролів. Подивимося, як виглядатиме в даному випадку протокол аутентифікації.

1. Аліса посилає системі свої ім'я і пароль.
2. Система обчислює односторонню функцію від пароля.
3. Система порівнює результат обчислення односторонньої функції зі значенням, що зберігається у базі даних облікових записів.

У результаті зменшуються втрати, які може задати зловмисник, коли отримає доступ до списку паролів, оскільки навіть у цьому разі за односторонньою функцією відновити паролі неможливо. Проте цей підхід не позбавлений недоліків.

Словникові атаки і сіль

Якщо зловмисник володіє списком паролів, зашифрованих односторонньою функцією, можлива словникова атака. Зловмисник бере набір найпоширеніших паролів, застосовує до них односторонню функцію і зберігає всі зашифровані паролі. Потім він порівнює список зашифрованих паролів із цим файлом (словником) у пошуках збігів.

Один зі способів боротьби із такою атакою пов'язаний із використанням *солі* (salt). Сіль — це випадковий рядок S, який додають до пароля перед шифруванням. У список шифрованих паролів заноситься рядок $S + E(S + P)$, де P — пароль, E — функція шифрування, «+» — конкатенація рядків. Якщо кількість можливих значень солі достатньо велика, то це робить словникову атаку значно складнішою, оскільки у словник потрібно вносити результати шифрування паролів із усіма можливими значеннями солі.

Солі потрібно досить багато. Наприклад, стандартний її обсяг, прийнятий в UNIX (12 біт, що дає 4096 можливих значень), є не зовсім достатнім (є словники найуживаніших паролів, об'єднані з усіма значеннями солі).

Аутентифікація за принципом «виклик-відповідь»

Більш серйозна проблема, пов'язана із використанням описаного підходу, полягає в тому, що пароль передають мережею незашифрованим, і він може бути перехоплений зловмисником. Один зі способів вирішення цієї проблеми полягає в тому, щоб передавати мережею не паролі, а їх односторонні хеші (дайджести). Цей підхід називають аутентифікацією за принципом «виклик-відповідь» (challenge-response authentication) або дайджест-аутентифікацією [32].

Протокол такої аутентифікації має такий вигляд.

1. Система зберігає значення односторонньої функції від пароля Аліси $F_1(P_S)$ у базі даних облікових записів.
2. Аліса передає системі своє вхідне ім'я (відкритим текстом) і значення $F_1(P_A)$, обчислене із використанням пароля P_A, який вона ввела.
3. Система генерує випадкове число C, яке називають *викликом* (challenge), і передає його Алісі.
4. Аліса застосовує іншу односторонню функцію до значення виклику. На вхід цієї функції, крім виклику, передають значення $F_1(P_A)$

$$R_A = F_2(F_1(P_A), C).$$

5. Аліса передає системі значення R_A (*відповідь*, response).
6. Система обчислює аналогічне до R_A значення R_S на підставі інформації із бази даних облікових записів

$$R_S = F_2(F_1(P_S), C).$$

7. Якщо значення R_A і R_S, отримані системою на кроках 5 і 6, збігаються, аутентифікацію вважають успішною.

Використання випадкового значення виклику в цьому разі зумовлене необхідністю запобігання *атаці відтворенням* (replay attack), під час якої зловмисник перехоплює інформацію, передану Алісою системі для того, щоб пізніше відіслати її самому, прикидаючись Алісою.

Цей протокол був основним підходом до аутентифікації у системах лінії Windows XP до появи Windows 2000 і дотепер підтримується у цих системах (наприклад, для локальної аутентифікації). Його надійність залежить від надійності алгоритму, використаного для односторонньої функції (перехоплення зловмисником дайджесту дасть можливість здійснити на нього словникову атаку). Зазначимо також, що в цьому разі користувач не може бути впевнений, що система насправді є тією, до якої він запитує доступ.

Більш складним протоколом аутентифікації є *протокол Kerberos* [51]. Це розподілена система аутентифікації користувачів із можливістю аутентифікації клієнта і сервера. Протокол Kerberos є основним протоколом мережної аутентифікації у системах лінії Windows XP, починаючи із Windows 2000. Реалізація цього протоколу доступна і для UNIX-систем.

Одноразові паролі

Проблему пересилання пароля мережею можна також розв'язати, використовуючи паролі, дійсні лише один раз під час сесії користувача. Перехоплення такого *одноразового пароля* (one-time password) нічого не дає зловмисникові.

Протокол використання одноразових паролів наведено нижче.

1. Аліса задає випадкове значення R для ініціалізації системи.
2. Система обчислює односторонні функції $F(R)$, $F(F(R))$, $F(F(F(...(R)))$, наприклад $n + 1$ раз. Назвемо ці числа $x_1 ... x_{n+1}$; із них $x_1 ... x_n$ система передає Алісі, а сама зберігає x_{n+1}.
3. Входячи в систему, Аліса задає своє ім'я і число x_n. Система обчислює $F(x_n)$ і порівнює результат із x_{n+1}. Якщо значення збігаються, аутентифікацію Аліси вважають успішною, і збережене системою x_{n+1} замінюється на x_n.
4. Для наступної аутентифікації Аліса використає число x_{n-1}, далі — x_{n-2} і т. д. Коли буде використане число x_1, систему повторно ініціалізують новим числом R.

Такі паролі у сучасних системах можуть реалізовуватися за допомогою смарт-карт — електронних пристроїв, у які вбудований мікропроцесор із засобами генерації відповідних одноразових паролів.

18.3.2. Основи керування доступом

Для реалізації керування доступом операційна система має можливість визначати, що за дії і над якими об'єктами має право виконувати той чи інший користувач системи. Звичайний розподіл прав відображають у вигляді *матриці доступу*, у якій рядки відповідають суб'єктам авторизації (користувачам, групам користувачів тощо), стовпці — ресурсам (файлам, пристроям тощо), а на перетині рядка і стовпця зазначені права цього суб'єкта на виконання операцій над даним ресурсом (рис. 18.1).

	Файл a.txt	Файл /bin/bash	Принтер 1
Ivanov	RW	RWX	W
Petrov	---	RWX	W
USERS	---	R	W

Рис. 18.1. Матриця доступу

Зберігання повної матриці контролю доступу неефективне (вона займатиме багато місця), тому звичайно використовують два базові підходи для її компактного відображення.

1. У разі реалізації *списків контролю доступу* (Access Control Lists, ACL) зберігаються стовпці матриці. Для кожного ресурсу задано список суб'єктів, які можуть використовувати цей ресурс.
2. У разі реалізації *можливостей* (capabilities) зберігаються рядки матриці. Для кожного суб'єкта задано список ресурсів, які йому дозволено використовувати.

Списки контролю доступу

Якщо реалізовано ACL, для кожного об'єкта задають список, що визначає, які користувачі можуть виконувати ті чи інші операції із цим об'єктом. Наприклад, для файлових систем такими об'єктами є файли або каталоги. У найзагальнішій формі елементи цього списку — це пари (користувач, набір дій), які називають *елементами контролю доступу* (access control elements, ACE).

Розглянемо деякі проблеми, які потрібно вирішити у разі реалізації списків контролю доступу.

- Розмір списку контролю доступу має бути не надто великим, щоб система могла ефективно ним керувати.
- Права мають бути досить гнучкими, але при цьому не дуже складними. Надмірне ускладнення системи прав звичайно призводить до того, що користувачі починають її «обходити», задаючи спрощені права; у результаті загальна безпека не підвищується, а ще більше знижується.

Вирішення цих проблем призвело до особливостей реалізації списків контролю доступу в різних ОС. Так, загальною концепцією у створенні списків є задання прав не лише для окремих користувачів, але й для груп користувачів, а також можливість визначити права доступу всіх користувачів системи. У деяких ОС обмежують кількість елементів у списку (наприклад, в UNIX список, як незабаром побачимо, обмежений трьома елементами). Водночас у системах лінії Windows XP можна задавати списки контролю доступу, що складаються з необмеженої кількості елементів (для окремих користувачів, груп, користувачів інших комп'ютерів тощо), і задавати різні комбінації прав — від узагальнених до детальних.

Можливості

Під час визначення можливостей для кожного користувача задають, до яких файлів він може мати доступ і як саме. При цьому із користувачем пов'язують *список можливостей* (capability list), що складається із пар (об'єкт, набір дій).

Реалізація списків можливостей дає змогу забезпечити не тільки захист, але й задання імен. Можна зробити так, щоб кожний користувач бачив тільки ті файли, до яких у нього є доступ. Якщо при цьому значенням за замовчуванням є відсутність доступу, користувачі можуть починати роботу в «порожній» системі. Загалом можливості використовують у системах, де потрібно забезпечити більшу безпеку за рахунок деякого зниження зручності роботи.

Можливості можуть бути задані не тільки для користувачів, але й для окремих процесів. У результаті під час запуску кожного процесу можна визначити його права.

18.4. Аутентифікація та керування доступом в UNIX

У цьому розділі опишемо, як реалізувати аутентифікацію і керувати доступом в UNIX-системах на прикладі Linux.

18.4.1. Облікові записи користувачів

Перш ніж розглянути реалізацію аутентифікації в UNIX, зупинимося на концепції користувача цієї системи.

Користувачі та групи користувачів

Кожному користувачу в UNIX ставлять у відповідність обліковий запис (account), що характеризується іменем користувача та ідентифікатором (uid). Як ім'я користувача, так і його ідентифікатор мають бути унікальними в межах усієї системи. Крім цього, з обліковим записом користувача пов'язують його *домашній каталог* (home directory), у який він за замовчуванням може записувати дані. Після входу користувача у систему відбувається перехід у його домашній каталог.

Користувачі об'єднуються в групи. Кожна група характеризується іменем та ідентифікатором (gid).

Загалом процес виконують із правами того користувача, який запустив відповідний виконуваний файл (винятки з цього правила наведемо у розділі 18.4.3).

Для доступу до ідентифікатора користувача поточного процесу використовують системний виклик getuid(), до ідентифікатора основної групи цього користувача — getgid()

```
#include <unistd.h>
printf("uid = %d, gid = %d\n", getuid(), getgid());
```

Суперкористувач root

Користувач із uid, що дорівнює нулю (звичайно його називають root) має в UNIX особливий статус — він може виконувати у системі будь-які дії без обмежень. Такого користувача називають також *суперкористувачем*. Усі інші — *звичайні користувачі*; про те, як задають права доступу для них, йтиметься в розділі 18.4.3.

Наявність єдиного суперкористувача (принцип «все або нічого») вважають головною слабкістю системи безпеки UNIX, оскільки компрометація єдиного пароля root негайно призводить до того, що зловмисник отримує повний контроль над системою. Крім цього, процеси під час виконання не можуть бути обмежені якоюсь частиною прав суперкористувача. Фактично, процес, якому потрібна лише невелика частина таких прав, змушений виконуватися під керуванням root із повним контролем над системою.

18.4.2. Аутентифікація

Для стандартної аутентифікації UNIX використовують підхід із використанням односторонньої функції від пароля і солі, описаний у пункті 18.3.1. Такою функцією традиційно був алгоритм DES, яким шифрувався рядок, що складався із нулів, при цьому в ролі ключа використовували значення пароля, об'єднане із сіллю. У сучасних версіях UNIX замість DES часто використовують односторонню хеш-функцію (звичайно MD5).

За вхід користувача у систему відповідають дві утиліти: getty — ініціалізує термінал, видає підказку «login:» і приймає ім'я користувача, і login, що видає підказку «password:», приймає значення пароля, робить аутентифікацію і починає сесію користувача (переходить у домашній каталог і запускає командний інтерпретатор).

Зберігання інформації про облікові записи

Інформацію про користувачів у UNIX зберігають у файлі /etc/passwd, який відіграє роль бази даних облікових записів. Кожному користувачу відповідає один рядок цього файла, що має таку структуру:

```
ім'я_користувача:шифрований_пароль:uid:gid:відомості:home_каталог:shell
```

де: ім'я_користувача — вхідне ім'я цього користувача;

шифрований_пароль — одностороння функція від пароля, обчислена із використанням DES, MD5 тощо (якщо значення цього поля починається з '*', користувач не може інтерактивно входити у систему);

uid і gid — чисельні ідентифікатори користувача та його основної групи (користувач може належати до кількох груп, інформація про це зберігається у файлі /etc/group, що розглядатиметься наступним);

відомості — додаткова інформація про користувача (його повне ім'я тощо);

home_каталог — місце знаходження домашнього каталогу користувача;

shell — версія командного інтерпретатора, який запускають для користувача, коли він входить у систему (у Linux це звичайно /bin/bash).

Наведемо приклад рядка /etc/passwd:

```
ivanov:10.elXw3GYJE:540:102:Іванов І.В.:/home/ivanov:/bin/bash
```

Інформацію про групи зберігають у файлі /etc/group. Рядок цього файла має синтаксис:

```
ім'я_групи:пароль_групи:gid:список_членів
```

де: gid — чисельний ідентифікатор групи (той самий, що в /etc/passwd);

список_членів — список імен усіх користувачів, які входять у групу, розділених комами.

Наведемо приклад рядка /etc/group:

```
students::102:ivanov,petrov,sidorov
```

Програмний доступ до облікових записів

Для доступу із застосувань до інформації, яку зберігають у /etc/passwd, можна використати функції, описані в <pwd.h>:

```
// доступ до інформації для заданого uid
struct passwd *getpwuid(uid_t uid);
// доступ до інформації для заданого імені користувача
struct passwd *getpwnam(char *name);
```

Обидві ці функції повертають покажчик на структуру passwd. У таблиці 18.1 наведена відповідність її полів елементам рядка файла /etc/passwd.

Таблиця 18.1. Відповідність полів структури passwd елементам рядка файла /etc/passwd

Поле структури passwd	Елемент рядка /etc/passwd
pw_name	ім'я_користувача
pw_passwd	шифрований_пароль
pw_gecos	відомості
pw_dir	home_каталог
pw_shell	shell

Усі наведені поля є рядками символів.

Приклад отримання інформації із цього файла наведено нижче.

```
struct passwd *pwd;
pwd = getpwuid(getuid());
printf("ім'я користувача: %s, домашній каталог: %s\n",
     pwd->pw_name, pwd->pw_dir);
```

Заздалегідь виділяти пам'ять під структуру passwd не потрібно — її розміщують у статичній пам'яті під час першого виклику однієї із наведених функцій. Наступні виклики використовують ту саму пам'ять, затираючи попередній результат, тому, якщо потрібно зберегти результат виклику, його копіюють в окрему пам'ять.

Тіньові паролі

У традиційних UNIX-системах до файла /etc/passwd мав доступ будь-який користувач ОС. А тому зловмисникові достатньо було отримати непривілейований доступ до системи, щоб почати словникову атаку на цей файл.

Для вирішення даної проблеми потрібно заборонити доступ до цього файла для всіх користувачів, крім root. Проте таке рішення не є прийнятним, оскільки в цьому файлі, крім інформації про паролі, містяться імена та ідентифікатори користувачів, розташування їхніх домашніх каталогів та інші дані, необхідні більшості застосувань. У результаті всі програми, яким потрібна така інформація, довелося б запускати із правами суперкористувача.

У сучасних UNIX-системах (зокрема в Linux) застосовують інший підхід — технологію *тіньових паролів* (shadow passwords). При цьому інформацію про паролі переносять із /etc/passwd в окремий тіньовий файл паролів (який зазвичай називають /etc/shadow). До нього може звертатися тільки суперкористувач. Іншу інформацію (імена та ідентифікатори користувачів тощо) зберігають у /etc/passwd, що залишається доступним для всіх користувачів. Тіньові паролі дають змогу значно підвищити надійність схеми аутентифікації системи.

Головною проблемою цього підходу є те, що в разі переходу до використання тіньових паролів усі прикладні програми, які працюють із паролями (login тощо), необхідно змінити та перекомпілювати. Найгнучкішим вирішенням у цьому разі є застосування *підключуваних модулів аутентифікації* (Pluggable Authentication Modules, PAM) [24], які дають можливість під час роботи системи визначати і налагоджувати процедуру аутентифікації для різних застосувань без зміни їхнього коду.

18.4.3. Керування доступом

Реалізація керування доступом до файлів у UNIX збереглася, майже не змінившись від ранніх версій системи. Фактично вона є скороченою реалізацією списків контролю доступу.

Основні принципи реалізації

Наведемо принципи реалізації керування доступом в UNIX.

1. Усі файли і каталоги в UNIX мають власника (owner), що належить до певної групи (group), і права доступу (permissions).
2. Розрізняють три категорії прав доступу: для читання, записування і виконання. Для звичайних файлів назви прав відповідають їхньому змісту (зазначимо,

що ядро UNIX намагатиметься завантажити у пам'ять і виконати будь-який файл, на який у поточного користувача є права на виконання).

3. Для каталогів задають той самий набір прав, але вони відрізняються за змістом від прав для файлів:
   - ✦ право читання для каталогу означає можливість отримання списку імен файлів, що містяться в ньому (тобто читання вмісту каталогу як файла);
   - ✦ право записування в каталог означає можливість створення в каталозі нових файлів і вилучення наявних (тобто записування в каталог як у файл); зазначимо, що для вилучення файла прав на нього самого можна й не мати — достатньо прав на каталог, де він зберігається;
   - ✦ право виконання означає можливість пошуку в каталозі, тобто доступу до окремих файлів.

   Із цього випливає, що якщо для каталогу задаються права «тільки для виконання», то доступ до окремих файлів у ньому можна отримати, коли знати їхні імена, список же всіх файлів отримати буде неможливо. Таке задання прав використовують, аби сторонні особи не могли випадково виявити файли, які, однак, потребують спільного доступу. З іншого боку, якщо задати права «тільки для читання», можна переглядати список файлів каталогу, але доступ до окремих файлів стане неможливий.
4. Права кожної категорії задають окремо для власника файла, для користувачів його групи і для всіх інших користувачів усього дев'ять базових комбінацій прав; можна сказати, що із кожним файлом пов'язаний список контролю доступу із трьох елементів.

Приклад задання прав

У UNIX-системах є стандартна утиліта виведення вмісту каталогу, яку називають `ls`. Ось фрагмент результату виклику цієї утиліти для каталогу

```
drwxr-xr-x   2 shekvl    osbook    1024  Sep 8 02:26 dir/
-rwxr-x---   1 shekvl    osbook   11370  Sep 3 13:18 test*
-rw-------   1 shekvl    osbook     149  Sep 3 13:41 test.c
```

Першим полем ліворуч є поле атрибутів файла, формат якого розглянемо докладніше. Перший стовпчик цього поля визначає тип файла (для каталогів у ньому стоятиме символ `d`, для звичайних файлів — прочерк, для символічних зв'язків — `l`).

Далі йде символьне відображення прав доступу. Воно складається із трьох трійок символів `rwx`. Ці трійки визначають (зліва направо): права власника, групи і всіх інших користувачів. У кожній трійці символ `r` означає право на читання, `w` — на записування, `x` — на виконання. Якщо відповідного права немає, то замість символу стоятиме прочерк.

У наведеному списку файлів:

- ✦ для каталогу `dir` задані усі права для власника, а також права на читання і виконання для групи та інших користувачів; це означає, що додавання і вилучення файлів дозволене власникові каталогу, інші дії — усім;

- виконуваний файл test може бути прочитаний і запущений власником і членами його групи, записаний — тільки власником, усі інші користувачі доступу до нього не мають;
- файл test.c може бути прочитаний і записаний тільки його власником, інші користувачі жодних прав на нього не мають.

Програмний інтерфейс роботи із правами доступу

Для зміни прав доступу до файла використовують системні виклики chmod() (файл задають іменем) і fchmod() (файл задають дескриптором), для зміни власника і групи — chown() і fchown(). Є також утиліти chmod, chown і chgrp.

Опишемо особливості використання виклику chmod():

```
int chmod(const char *pathname, int mode);
```

Параметр mode визначає права доступу. Вони можуть бути компактно записані у вісімковому числовому відображенні. Елемент списку контролю доступу відображають сумою трьох чисел: 1 — для права на виконання, 2 — на записування, 4 — на читання. Тому, наприклад, 0 означає відсутність прав, 5 — читання і виконання, 6 — читання і записування, 7 — усі права. Увесь список відображають сукупністю трьох чисел (у тому самому порядку, що і для символьного відображення); так, права для dir у цьому випадку були б задані як 755, для test — 750, для test.c — 600.

Ось приклад використання виклику chmod() для задання прав, що визначають можливість читання і записування для власника, читання для групи і відсутність доступу для інших користувачів:

```
chmod("./test.c", 0640);
```

Як було видно раніше, таке саме числове відображення може бути передане як режим створення файла у системні виклики open(), mkfifo() тощо:

```
int fd = open("./myfile.txt", O_RDWR | O_CREAT, 0640);
```

Для отримання прав доступу до заданого файла необхідно скористатися системними викликами stat() або fstat():

```
struct stat attrs;
stat("./myfile.txt", &attrs);
// S_IFMT задає маску, що визначає всю інформацію, крім прав доступу
printf("права доступу: %o\n", attrs.st_mode & ~S_IFMT);
```

Для перевірки того, чи є в поточного процесу права на доступ до заданого файла, використовують системний виклик access():

```
int access(const char *pathname, int mode);
```

де: pathname — повний шлях до файла;

mode — тип перевірки (R_OK — наявність права на читання, W_OK — на записування, X_OK — на виконання, F_OK — перевірка на наявність файла).

Виклик access() повертає 0 за наявності прав, –1 — за їхньої відсутності.

```
if (access("myfile.txt", R_OK) == -1)
    printf("немає права на читання файла\n");
```

Недоліки схеми керування доступом UNIX

Описана схема керування доступом проста для розуміння, але багато в чому обмежена. Основне обмеження пов'язане із довжиною списку контролю доступу (3 елементи). Звідси виникають такі проблеми, як неможливість надання конкретному користувачу прав доступу до файла без створення для нього окремої групи (що не можливо зробити без наявності прав адміністратора).

Останнім часом в UNIX-системах усе ширше використовують списки контролю доступу без обмеження кількості елементів (у Linux їхня реалізація стала доступна у ядрі 2.6).

Виконання із правами власника

Права доступу для файлів і каталогів у UNIX дають змогу задавати додаткові біти. Одним із таких бітів є setuid-біт. Для його задання у числовому зображенні ліворуч потрібно додати 4 і обов'язково задати право виконання для власника (наприклад, задати права як 4755). Символьне відображення в цьому разі має такий вигляд: `rwsr-xr-x`.

Setuid-біт задає для файла режим виконання із правами власника. Під час запуску такого файла на виконання створюють setuid-процес, для якого `uid` відповідає користувачу, що запустив програму, а `euid` – власникові виконуваного файла. Ядро ОС під час авторизації доступу до файлів завжди використовує `euid`. У результаті маємо, що setuid-процес отримує під час доступу до файлів такі самі права, що і власник відповідного виконуваного файлу.

Прикладом використання такого біта є програма `passwd`, що дає змогу користувачу задати пароль на вхід у систему. З одного боку, вона має бути доступна для запуску будь-якому користувачу системи, з іншого – під час зміни пароля їй потрібно перезаписувати файли `/etc/passwd` або `/etc/shadow`, що вимагає виконання із правами `root`. Щоб задовольнити ці вимоги, для виконуваного файла `passwd` власником вказують `root` і задають setuid-біт.

Для визначення ефективного ідентифікатора користувача для поточного процесу використовують системний виклик `geteuid()`:

```
// для setuid-програми uid і euid можуть розрізнятися
printf("uid=%d, euid=%d\n", getuid(), geteuid());
```

Застосування, які виконуються із правами `root`, можуть переходити до виконання із правами інших користувачів за допомогою системного виклику `setuid()`, що змінює значення `uid` і `euid` для поточного процесу:

```
struct passwd *pw = getpwnam("nobody");
setuid(pw->pw_uid);
// тепер процес виконується
// з правами користувача nobody
```

Таку технологію, наприклад, використовують у веб-сервері Apache, основний процес якого запускають із правами `root`, після чого він виконує підготовчі дії, а потім створює нащадків для обслуговування запитів, які негайно переходять до виконання із правами непривілейованого користувача.

Задання параметра umask

Залишається з'ясувати, які права задають для нових файлів. Цим керує встановлення прав за замовчуванням. Якщо його не змінювати, каталоги створюватимуть

із правами 777 (усі дії дозволені всім), а файли — із правами 666 (усі дії, крім виконання, дозволені всім).

Змінити встановлення прав можна, задаючи спеціальний *параметр umask* за допомогою системного виклику `umask()` або однойменної утиліти. Під час створення нового файла значення цього параметра побітово відніматиметься від 666 (вісімкового) для звичайних файлів і з 777 — для каталогів; права для файла будуть результатом цього віднімання. Наведемо приклад виклику `umask()`:

```
umask(022);
```

Після виконання цього виклику файли створюються із правами 644, а каталоги — із правами 755. Найчастіше задають значення umask, що дорівнюють 022 і 002, задання 077 спричинятиме створення файлів, закритих для всіх, крім власника.

Підтримка можливостей у Linux

Ядро Linux дає змогу задавати списки можливостей для процесів, починаючи з версії 2.4. Структура таких списків аналогічна до описаної у розділі 18.3.2.

Наявність підтримки можливостей дозволяє обійти обмеження концепції «все або нічого», коли процесу потрібна тільки частина прав суперкористувача і він змушений брати їх усі, ризикуючи нашкодити системі. Тепер такий процес може задати набір можливостей, що точно описує права, які йому потрібні.

18.5. Аутентифікація і керування доступом у Windows XP

18.5.1. Загальна архітектура безпеки

Архітектури безпеки Windows XP [14] містять такі основні компоненти.

- *Процес реєстрації користувачів* (logon process, winlogon) обробляє запити користувачів на реєстрацію у системі та приймає дані від них.
- *Менеджер аутентифікації* (Local Security Authority Subsystem, LSASS) безпосередньо проводить аутентифікацію користувача. Цей компонент — центральний у підсистемі безпеки. Крім аутентифікації, він контролює політику аудиту, про яку йтиметься у розділі 18.6.
- *Менеджер облікових записів* (Security Accounts Manager, SAM) підтримує базу даних облікових записів (базу даних SAM), що містить імена локальних користувачів і груп, а також паролі. Під час перевірки прав користувача SAM взаємодіє із менеджером аутентифікації.
- *Довідковий монітор захисту* (Security Reference Monitor, SRM) перевіряє права користувача на доступ до об'єкта і виконує необхідну дію з об'єктом за наявності цих прав. Це єдиний компонент, що виконується в режимі ядра, він реалізує політику контролю доступу, визначену менеджером аутентифікації. SRM гарантує, що будь-який користувач або процес, що дістав доступ до об'єкта, має всі права на нього.

Ця архітектура реалізована у системах, що не використовують мережної аутентифікації. Для неї інформація про облікові записи має зберігатися не у базі

даних SAM, а у спеціальному централізованому сховищі даних — *активному каталозі* (Active Directory). Під час мережної аутентифікації менеджер аутентифікації звертається до служби активного каталогу віддаленого комп'ютера, на якому розташований цей каталог. База даних SAM, однак, є у будь-якій установці ОС лінії Windows XP — у ній зберігають облікові записи для локальної аутентифікації.

Подальший виклад торкатиметься локальної аутентифікації.

18.5.2. Аутентифікація

Windows XP вимагає, щоб кожному користувачу відповідав обліковий запис. Він пов'язаний із *профілем захисту*, який є набором інформації щодо контролю доступу (ім'я користувача, список його груп, пароль тощо). Профілі захисту зберігають у базі даних SAM і використовують для аутентифікації.

Розглянемо послідовність кроків реєстрації користувача у системі. Процес winlogon очікує введення від користувача. Потік цього процесу виявляє спробу користувача ввійти у систему (натискання комбінації клавіш Ctrl+Alt+Del) і пропонує йому ввести ім'я облікового запису та пароль. При цьому для реалізації такого введення winlogon звертається до спеціальної DLL графічної ідентифікації та аутентифікації (GINA). Стандартне вікно введення даних користувача реалізоване у msgina.dll, програміст може встановити свою версію цієї динамічної бібліотеки, що реалізує альтернативний метод аутентифікації (наприклад, на основі смарт-карт або біометричних даних).

Дані від користувача передаються процесу winlogon, і аутентифікація із використанням бази даних SAM (за яку відповідає LSASS) відбувається відповідно до протоколу «виклик-відповідь». Можна використати і протокол Керберос (цей підхід тут не розглядатиметься).

Якщо аутентифікація пройшла успішно, створюють об'єкт, що унікальним чином визначає цього користувача в усіх його подальших діях. Цей об'єкт, який називають *маркером доступу* (access token), відіграє ключову роль у підсистемі захисту: він визначає, до яких системних ресурсів мають доступ потоки, створені цим користувачем.

Після успішної аутентифікації користувача LSASS створює процес і приєднує до нього маркер доступу користувача. Цей процес передають підсистемі Win32, що запускає в його адресному просторі застосування, визначене у реєстрі як оболонка системи (за замовчуванням ним є стандартна оболонка explorer.exe). Застосування формує візуальне відображення параметрів робочого столу для користувача.

Ідентифікатори безпеки

Із кожним користувачем і групою у Windows XP пов'язують унікальний *ідентифікатор безпеки* (Security Identifier, SID). Це цілочислове значення, що складається із заголовка і випадкової частини. Система безпеки звертається до користувачів і груп тільки за їхнім SID.

Win32 API надає ряд функцій для роботи із SID [32]. Для отримання SID користувача або групи за іменем використовують функцію LookupAccountName():

```
BOOL LookupAccountName(LPCTSTR sname, LPCTSTR username, PSID sid,
  LPDWORD sidsize, LPTSTR dname, LPDWORD dsize, PSID_NAME_USE sidtype);
```

де: username — ім'я користувача;

sid — покажчик на буфер пам'яті, у який записується отриманий SID, відображений спеціальною структурою SID (цей буфер звичайно розміщують у динамічній ділянці пам'яті);

sidsize — розмір буфера (якщо його недостатньо для розміщення SID, у дану змінну записується необхідний розмір, а буфер sid не заповнюється);

dname — покажчик на буфер пам'яті, у який записується ім'я домена, де зареєстрований користувач (для локального користувача — ім'я комп'ютера), dsize — розмір цього буфера;

sidtype — покажчик на змінну, що задає тип SID (вона може набувати значень SidTypeUser — SID користувача, SidTypeGroup — SID групи).

Наведемо приклад отримання SID користувача за його ім'ям:

```
DWORD sidsize = 1024, domsize = 1024;
char username[] = "ivanov", domname[1024];
SID_NAME_USE sidtype = SidTypeUser;
PSID sid = (PSID)HeapAlloc(GetProcessHeap(), 0, sidsize);
LookupAccountName(NULL, username, sid, &sidsize, domname,
    &domsize, &sidtype);
// ... робота з sid
FreeSid(sid);
```

Для перетворення структури SID у рядковий формат, придатний для відображення, використовують функцію ConvertSidToStringSid():

```
#include <sddl.h>
char *ssid = NULL;
// PSID sid; LookupAccountName(... sid, ...);
ConvertSidToStringSid(sid, &ssid);
// пам'ять для ssid буде виділено автоматично
printf("%s\n", ssid); // S-1-5-21-1844237615-1682526488-1202660629-1002
LocalFree(ssid);
```

Маркери доступу

Маркер доступу користувача пов'язують з усіма процесами, які він створює. Маркер є «посвідченням особи» процесу, коли той намагається використати який-небудь системний ресурс, і містить таку інформацію:

- SID користувача, із правами якого виконується процес;
- список груп, до яких належить цей користувач;
- список привілеїв, якими володіє користувач;
- список контролю доступу за замовчуванням, який визначає первісні права доступу для об'єктів, створюваних процесами цього користувача (про структуру такого списку йтиметься у пункті 18.5.3).

У разі спроби процесу відкрити дескриптор об'єкта диспетчер об'єктів звертається до SRM. SRM отримує маркер доступу, пов'язаний із процесом, та використовує його SID і список груп, щоб визначити, чи має процес право доступу до об'єкта. Фактично маркер доступу відповідає ідентифікатору euid для UNIX.

Для отримання маркера доступу процесу використовують функцію OpenProcessToken():

```
BOOL OpenProcessToken(HANDLE ph, DWORD access, PHANDLE ptoken);
```

де: ph — дескриптор процесу;

access — тип доступу до маркера (TOKEN_QUERY — читання інформації);

ptoken — покажчик на змінну, в яку записується дескриптор маркера доступу.

Ось приклад отримання маркера доступу поточного процесу:

```
HANDLE mytoken;
OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &mytoken);
```

Інформація, що зберігається у маркері доступу, може бути отримана за допомогою функції GetTokenInformation():

```
BOOL GetTokenInformation(HANDLE token, TOKEN_INFORMATION_CLASS tclass,
 LPVOID tibuf, DWORD tisize, PDWORD realsize);
```

де: token — дескриптор маркера доступу;

tclass — значення, що визначає тип отримуваної інформації (TokenUser — інформація про обліковий запис користувача, TokenGroups — список груп);

tibuf — буфер для розміщення інформації (якщо tclass дорівнює TokenUser, цей буфер містить структуру типу TOKEN_USER із полем User, що відображає структуру із полем Sid, яке містить SID цього користувача);

tisize — розмір буфера;

realsize — покажчик на змінну, що містить розмір буфера, необхідного для розміщення інформації (якщо tisize недостатньо, функція поверне FALSE і запише в цю змінну потрібний розмір).

Наведемо приклад отримання SID з маркера доступу:

```
GetTokenInformation(mytoken, TokenUser, buf, sizeof(buf), &size);
ConvertSidToStringSid(((PTOKEN_USER)buf)->User.Sid, &ssid);
printf("%s\n", ssid);
```

Запозичення прав

Окремі потоки процесу можуть виконуватися із правами користувачів, які відрізняються від прав творця цього процесу. Такий підхід називають *запозиченням прав* (impersonation), за своїм призначенням він відповідає системному виклику setuid() в UNIX-системах. Як і setuid(), запозичення прав найчастіше використовують у серверах, які виконує користувач із адміністративними привілеями. У цьому разі потоки, що обслуговують запити користувачів, можуть виконуватися із меншими правами.

Потік, для якого задано запозичення прав, отримує окремий маркер доступу, який називають *маркером режиму запозичення прав* (impersonation token). Подальшу перевірку прав доступу виконують із використанням цього маркера.

Для отримання маркера доступу довільного користувача необхідно здійснити програмний вхід у систему для цього користувача. Такий вхід реалізує функція LogonUser():

```
BOOL LogonUser(LPTSTR username, LPTSTR domain, LPTSTR passwd,
   DWORD logon_type, DWORD logon_provider, PHANDLE ptoken);
```

де: username — ім'я користувача, domain — домен або комп'ютер, у якому зареєстрований користувач (для локальної системи задають "."), passwd — пароль користувача;

logon_type — тип входу у систему (для звичайного користувача потрібно задавати LOGON32_LOGON_INTERACTIVE);

logon_provider — алгоритм аутентифікації, який використовують для входу (LOGON32_PROVIDER_DEFAULT — стандартний алгоритм);

ptoken — покажчик на змінну, у яку поміщається маркер доступу цього користувача.

Наведемо приклад коду, що здійснює програмний вхід у систему:

```
HANDLE itoken;
LogonUser("ivanov", ".", "ivanov_pwd", LOGON32_LOGON_INTERACTIVE,
   LOGON32_PROVIDER_DEFAULT, &itoken);
```

Реалізацію запозичення прав у користувача, що присутній у системі, здійснюють за допомогою функції ImpersonateLoggedOnUser():

```
ImpersonateLoggedOnUser(itoken);
// ... потік виконується з правами користувача, що володіє itoken
```

Маркер доступу потоку може бути отриманий за допомогою функції OpenThreadToken(), аналогічної до OpenProcessToken():

```
OpenThreadToken(GetCurrentThread(), TOKEN_QUERY, TRUE, &ttoken);
// ... GetTokenInformation(ttoken, ...) і т. д.
```

Для повернення до виконання із колишніми правами використовують функцію RevertToSelf():

```
RevertToSelf();
```

18.5.3. Керування доступом

Під час створення будь-якого об'єкта Windows XP, який може бути використаний більш як одним процесом (включаючи файли, поіменовані канали, синхронізаційні об'єкти тощо), йому присвоюють *дескриптор захисту* (security descriptor).

До найважливіших елементів дескриптора захисту належать:

- SID власника об'єкта (власник завжди може змінювати атрибути безпеки об'єкта, навіть якщо в нього немає прав на доступ до його даних);
- список контролю доступу (ACL), що визначає права доступу до об'єкта.

Кожний елемент списку контролю доступу (ACE) містить такі елементи:

- тип ACE (виділяють, зокрема, дозволяючі і забороняючі ACE);
- ідентифікатор безпеки (SID);
- набір прав доступу (читання, записування, повний контроль тощо).

Сума прав доступу, наданих окремими ACE, формує загальний набір прав доступу, наданих ACL.

Розглянемо спрощений приклад ACL для файлового об'єкта (рис. 18.2).

Якщо файловий об'єкт володіє таким ACL, користувач може читати з відповідного файла, коли:

- під час пошуку в ACL файлового об'єкта знайдено ACE, що містить SID цього користувача або однієї із груп, куди він входить;
- цей ACE дозволяє доступ;
- у ньому є право на читання даних.

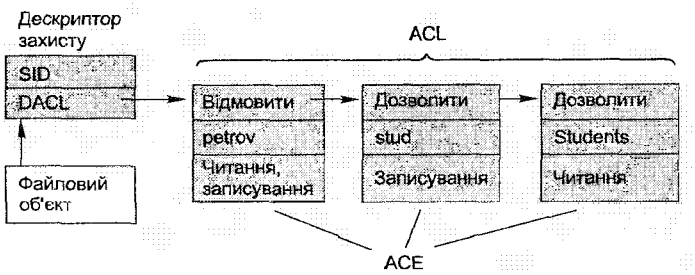

Рис. 18.2. Список контролю доступу

Щоб визначити, який ACL має призначатися новому об'єкту, система захисту застосовує три правила в такому порядку.

1. Якщо ACL явно задано під час створення об'єкта, то система захисту присвоює його об'єкту.
2. Якщо ACL не задано, і в об'єкта є ім'я, система захисту шукає ACL каталогу об'єктів, де збережеться ім'я цього об'єкта (для файла таким каталогом буде каталог файлової системи). Деякі ACE каталогу можуть бути позначені як «успадковувані». Це означає, що вони присвоюються новим об'єктам, створеним у цьому каталозі. Якщо такі успадковувані ACE є, із них складають ACL, що призначають новому об'єкту.
3. Якщо жоден із перших двох випадків не стався, об'єкту присвоюють ACL за замовчуванням із маркера доступу процесу, що робив виклик.

Використання списків контролю доступу

Коли потік намагається відкрити дескриптор об'єкта, диспетчер об'єктів і підсистема захисту зіставляють інформацію із маркера доступу відповідного процесу (що визначає власника процесу) і дескриптора захисту об'єкта (що визначає користувачів, які можуть мати доступ до об'єкта), щоб встановити, чи можна надати потоку запитаний дескриптор.

Подивимося, що відбувається, коли потік, запущений користувачем stud (що входить у групу Students), відкриває файл для читання, викликавши функцію

```
CreateFile("myfile.txt", GENERIC_READ, ...);
```

У цьому разі підсистема безпеки переглядає список контролю доступу для файла myfile.txt, починаючи з першого ACE. Якщо під час перегляду в одному з ACE цього списку виявляють SID потоку, що робив виклик (узятий із його маркера доступу, в нашому випадку це буде SID для stud), або його групи, перегляд зупиняють і перевіряють, чи дозволяє цей ACE доступ для читання. Коли так, пошук зупиняють, і `CreateFile()` повертає дескриптор файла. Якщо SID користувача або групи не знайдено (або знайдено в ACE із забороняючим доступом), `CreateFile()` поверне помилку.

Елементи, що забороняють доступ, поміщають у початок ACL. Інакше виникла б можлива ситуація, коли дозволяючий ACE з SID-групи, куди входить користувач, перебував перед забороняючим ACE з SID власне цього користувача, внаслідок чого той отримував би доступ до об'єкта, незважаючи на явну заборону.

Якщо ACL для дескриптора безпеки заданий, але не містить жодного елемента, ніхто із користувачів системи не має прав на роботу із цим об'єктом. Якщо ж ACL не заданий (дорівнює NULL), це означає, що будь-який користувач може виконувати будь-які дії з об'єктом. Це становить загрозу безпеці системи, оскільки за такої ситуації зловмисник може змінити права на цей об'єкт, наприклад закрити його для читання від усіх, вивівши цим із ладу застосування, що його використовують.

Задання прав доступу під час створення файла

Система прав доступу в ОС лінії Windows XP досить складна для розуміння і використання. Далі розглянемо простий приклад задання прав доступу для читання і записування під час створення файла [32, 50].

На першому етапі необхідно отримати доступ до SID, які заносяться у список контролю доступу. Для цього можна використати функцію LookupAccountName().

Далі потрібно розмістити у пам'яті та проініціалізувати список контролю доступу. Для цього треба розрахувати обсяг пам'яті, необхідний для його зберігання, і виділити пам'ять у динамічній ділянці. Цей обсяг отримують додаванням розмірів усіх ACE (що включають розміри ACE-структури і SID) і розміру ACL. Розмір SID повертає функція GetLengthSid():

```
// PSID ivanov_sid; LookupAccountName("ivanov", ... ivanov_sid, ...);
// два дозволяючих ACE – для читання і записування
DWORD acl_size = 2 * (GetLengthSid(ivanov_sid) +
    sizeof(ACCESS_ALLOWED_ACE)) + sizeof(ACL) ;
PACL pacl = (PACL)HeapAlloc(GetProcessHeap(), 0, acl_size);
```

Після виділення пам'яті її треба проініціалізувати за допомогою функції InitializeAcl(). Першим параметром вона приймає покажчик на виділену пам'ять, другим – її обсяг у байтах

```
InitializeAcl(pacl, acl_size, ACL_REVISION);
```

Кожний елемент контролю доступу потрібно додати в ACL. Для створення і додавання в ACL дозволяючих елементів використовують функцію AccessAllowedAce(), заборонних – AccessDeniedAce():

```
BOOL AddAccessAllowedAce(PACL pacl, DWORD rev, DWORD amask, PSID pSid);
```

де: pacl – покажчик на ACL;

amask – маска прапорців прав доступу (GENERIC_READ, GENERIC_WRITE тощо, повні права задають як GENERIC_ALL);

psid – покажчик на SID, для якого цей елемент задає права.

Ось приклад використання цих двох функцій:

```
// доступ для читання
AddAccessAllowedAce(pacl, ACL_REVISION, GENERIC_READ, ivanov_sid);
// доступ для записування
AddAccessAllowedAce(pacl, ACL_REVISION, GENERIC_WRITE, ivanov_sid);
```

На цьому етапі потрібний ACL повністю сформовано. Тепер необхідно сформувати дескриптор безпеки. Для цього слід розмістити у пам'яті структуру типу SECURITY_DESCRIPTOR і проініціалізувати її за допомогою функції InitializeSecurityDescriptor():

```
SECURITY_DESCRIPTOR sd;
InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION);
```

Далі в цю структуру додають ACL за допомогою функції SetSecurityDescriptorDacl():

```
BOOL SetSecurityDescriptorDacl(PSECURITY_DESCRIPTOR psd,
    BOOL dacl_present, PACL pdacl, BOOL dacl_from_default);
```

де: psd — покажчик на дескриптор безпеки, для якого задають ACL;

dacl_present — TRUE, якщо ACL задають, FALSE — якщо очищують;

pdacl — покажчик на ACL.

Наведемо код, який додає ACL у дескриптор безпеки.

```
SetSecurityDescriptorDacl(&sd, TRUE, pacl, FALSE);
```

Після створення дескриптора безпеки його потрібно передати у функцію створення файла. Як уже було видно, у цю функцію (і в багато інших) як параметр передають покажчик на структуру SECURITY_ATTRIBUTES. Покажчик на дескриптор безпеки задано як поле lpSecurityDescriptor цієї структури.

```
SECURITY_ATTRIBUTES sa = { 0 };
sa.nLength = sizeof(sa) ;
sa.lpSecurityDescriptor = &sd;
HANDLE hf = CreateFile("f:\\test.txt", GENERIC_READ | GENERIC_WRITE, 0,
    &sa, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

Після завершення роботи потрібно вивільнити пам'ять, виділену для структур ACL і SID, а також закрити дескриптор файла:

```
HeapFree(GetProcessHeap(), 0, pacl);
FreeSid(ivanov_sid);
CloseHandle(hf);
```

18.6. Аудит

У цьому розділі йтиметься про основні компоненти підсистеми аудиту сучасних ОС і особливості генерації повідомлень аудиту в застосуваннях користувача на прикладах Windows XP і UNIX.

18.6.1. Загальні принципи організації аудиту

Підсистеми аудиту сучасних ОС містять такі основні компоненти.

- Засоби визначення *політики аудиту* (audit policy). Визначивши її, адміністратор системи може задати перелік подій, які його цікавлять. Для зручності визначення політики повідомлення поділяються на рівні залежно від важливості.
- Засоби генерації повідомлень аудиту. Такі повідомлення можуть бути згенеровані різними компонентами режиму ядра і режиму користувача. Для цього ОС має надавати відповідну функціональність і системні виклики.
- Засоби збереження інформації в журналі аудиту. Їх зазвичай реалізують централізовано в рамках окремого комп'ютера або мережі, для чого запускають спеціальний фоновий процес, якому передають усю інформацію про повідомлення аудиту.

Треба зазначити, що зловмисник може змінити інформацію в журналі аудиту, щоб приховати сліди свого перебування у системі. Для запобігання таким змінам

у системах із високим ступенем захисту журнал аудиту можуть зберігати на віддаленому комп'ютері або на альтернативному носії (наприклад, відразу виводити на принтер або записувати на CD-ROM). Крім того, можна передбачити різні візуальні та звукові сигнали тривоги, що привертають увагу адміністратора.

18.6.2. Робота із системним журналом UNIX

Організація системного журналу

Для організації аудиту в UNIX-подібних системах використовують *системний журнал* (system log, syslog). За реєстрацію інформації від застосувань у ньому відповідає фоновий процес `syslogd`. Звичайно інформацію записують у файли, що перебувають у каталозі **/var/log** (керують процесом аудиту зміною конфігураційного файла **/etc/syslog.conf**). Усі системні фонові процеси і засоби забезпечення безпеки (наприклад, `login`) зберігають інформацію про свою роботу в цьому журналі. Адміністратори системи мають регулярно переглядати відповідні файли в пошуках можливих порушень політики безпеки.

Програмний інтерфейс системного журналу

Прикладні програми також можуть зберігати інформацію про свою діяльність у журналі за допомогою відповідного програмного інтерфейсу [24].

Зв'язок між процесом користувача і `syslogd` здійснюють через спеціальний сокет. Перед початком використання журналу цей сокет можна відкрити за допомогою системного виклику `openlog()`:

```
#include <syslog.h>
void openlog(char *ident, int options, int facility);
```

де: `ident` — рядок символів, що зберігатиметься в кожному записі журналу (звичайно це назва застосування);

`options` — маска прапорців, що визначають додаткові параметри використання журналу (`LOG_PID` — кожний запис журналу міститиме ідентифікатор процесу, `LOG_CONS` — повідомлення виводяться на консоль у разі неможливості виведення в журнал).

Наведемо код, що відкриває сокет журналу:

```
openlog("myserver", LOG_PID | LOG_CONS, 0);
```

Використання `openlog()` не є обов'язковим (якщо його не виконати, сокет журналу відкривається під час першої спроби записування в нього), але тільки так задають рядок, за яким можна буде розрізняти повідомлення різних процесів.

Записи в журнал здійснюють за допомогою функції `syslog()`:

```
syslog(int level, char *format, ...);
```

де: `level` — рівень серйозності повідомлення (`LOG_INFO` — інформаційне повідомлення, `LOG_WARNING` — попередження, `LOG_ERR` — повідомлення про помилку);

`format` — формат виведення повідомлення (взаємодія цього параметра і наступних аналогічна до `printf()`, за винятком того, що можливий додатковий модифікатор `%m`, який викликає відображення рядка повідомлення про помилку, аналогічного до `perror()`).

Ось як здійснюється реєстрація записів у журналі:

```
syslog(LOG_INFO, "процес запущено, euid=%d", geteuid());
syslog(LOG_WARNING, "неможливо відкрити файл %s: %m", fname);
```

Повідомлення різного рівня серйозності можуть зберігатися в різних файлах. Наприклад, у багатьох Linux-системах задання рівня LOG_ERR призводить до записування повідомлення у файл /var/log/syslog, а повідомлення нижчих рівнів — у файл /var/log/messages.

Наведемо приклад записів у журналі у разі використання прапорця LOG_PID

```
Oct  4 12:44:17 myhost myserver[2074]: процес запущено, euid=0
Oct 14 18:29:31 myhost myserver[2074]: неможливо відкрити файл a.txt:
No such file or directory
```

Після закінчення роботи із журналом його можна закрити (відповідний виклик закриває сокет, виділений для зв'язку із процесом syslogd):

```
closelog();
```

18.6.3. Журнал подій Windows XP

Архітектура аудиту Windows XP

Повідомлення аудиту можуть бути згенеровані виконавчою підсистемою під час перевірки прав доступу, при цьому за передавання таких повідомлень у режим користувача для реєстрації відповідає SRM. Крім того, повідомлення можуть надходити від компонентів режиму користувача, зокрема від прикладних процесів (для цього у програмах необхідно використовувати відповідні функції Win32 API). Інформацію про такі повідомлення зберігають у *журналі подій* (event log). Для роботи із ним процес повинен мати необхідні привілеї.

Необхідність аудиту конкретної події визначає *локальна політика безпеки* (local security policy). Цю політику підтримує LSASS у межах загальної політики безпеки і передає SRM під час ініціалізації системи. LSASS отримує повідомлення аудиту від SRM або компонентів режиму користувача і відсилає їх окремому процесові журналу подій, що зберігає записи аудиту в журналі.

Використання журналу подій у застосуваннях

Розглянемо особливості використання журналу подій у застосуваннях [32]. Насамперед необхідно зазначити, що процедура виведення повідомлень у журнал відокремлена від процедури задання тексту повідомлень. Фактично під час виведення повідомлення зазначають лише його код, а всю текстову інформацію, що відповідає цьому коду, задають в окремих файлах повідомлень журналу. Спочатку ознайомимося із процесом створення файлів повідомлень, а потім перейдемо до організації виведення повідомлень у журнал.

Розробка файлів повідомлень

Вихідні файли повідомлень — це текстові файли спеціального формату, які мають розширення .mc. Структура такого файла проста — він складається з пар ім'я_параметра=значення.

Журнал подій дає змогу задавати повідомлення кількома мовами. При цьому вибір варіанта повідомлення, що відображатиметься під час перегляду журналу

за допомогою вікна перегляду подій (Event Viewer), залежить від мовних налаштувань ОС. Для визначення допустимих мов на початку файла повідомлень необхідно задати параметр LanguageNames зі значенням у вигляді списку специфікацій мов, розділених пробілами (у форматі ім'я_мови=код_мови:ім'я_файла):

```
LanguageNames=(English=0x409:msg_en Ukrainian=0x422:msg_ua)
```

Блок інформації для окремого повідомлення має такий вигляд:

```
MessageId=0x1
SymbolicName=MSG_MYMSG
Language=English
Application started (executable file: %1)
.
Language=Ukrainian
Програму запущено (виконується файл: %1)
.
```

Параметр MessageId задає код повідомлення, SymbolicName визначає символьну константу, яка буде доступна в застосуванні як позначення цього повідомлення. Параметр Language починає блок тексту повідомлення, що відповідає заданій мові (ім'я мови треба задати раніше за допомогою параметра LanguageNames). Завершенням такого блоку є рядок із єдиним символом ".". У тексті повідомлення можливі специфікації підстановки %1, %2 тощо, замість них підставляються параметри, задані під час виведення повідомлення.

Після створення mc-файл обробляє компілятор повідомлень (mc.exe).

```
mc -A msgfile.mc
```

Параметр -A означає, що вихідні дані, як і вхідні, будуть у форматі ASCII.

У результаті буде згенеровано такі файли:

- заголовний файл msgfile.h із оголошеннями констант, заданих параметрами SymbolicName;
- набір файлів із визначеннями текстів для різних мов (наприклад, під час компіляції msgfile.mc будуть згенеровані msg_en.bin та msg_ua.bin);
- файл ресурсів msgfile.rc, що містить посилання на файли із визначеннями текстів.

Після компіляції mc-файла потрібно створити двійковий файл повідомлень. Звичайно його компонують у DLL за два етапи:

- текстовий файл ресурсів компілюють у двійкове подання за допомогою компілятора ресурсів

  ```
  rc -r -fo msgfile.res msgfile.rc
  ```

- на основі двійкового подання ресурсів компонують динамічну бібліотеку

  ```
  link -dll -noentry -out:msgfile.dll msgfile.res
  ```

Реєстрація двійкового файла повідомлень у реєстрі

Файл повідомлень повинен реєструватися у системному реєстрі. Для цього необхідно задати ключ:

```
HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
   Application\ім'я_джерела_повідомлень
```

і в ньому — рядкове значення EventMessageFile, яке повинно містити повний шлях до двійкового файла повідомлень. Ім'я джерела повідомлень з'являтиметься у відповідному стовпчику вікна перегляду повідомлень.

Після реєстрації повідомлення, пов'язані із джерелом даних, будуть відображені із використанням зазначеного файла повідомлень.

Виведення повідомлень у журнал

Після підготовки і реєстрації у системі двійкового файла повідомлень застосування може працювати із журналом подій.

Для виведення повідомлень у журнал використовують їхні коди, у ролі яких задають константи з отриманого раніше заголовного файла. Цей файл потрібно підключати до застосувань, що використовують журнал

```
#include "msgfile.h"
```

Перед виведенням повідомлень у журнал необхідно зареєструвати у системі джерело повідомлень, після завершення виведення — скасувати цю реєстрацію. Для цього використовують функції RegisterEventSource() і DeregisterEventSource():

```
// myapp повинен збігатися з ім'ям джерела, заданим у реєстрі
HANDLE hlog = RegisterEventSource(NULL,"myapp");
// ... робота з журналом через дескриптор hlog
DeregisterEventSource(hlog);
```

Для виведення повідомлення в журнал використовують функцію ReportEvent():

```
BOOL ReportEvent(HANDLE hlog, WORD etype, WORD ecategory, DWORD ecode,
  PSID userid, WORD parnum, DWORD dsize, LPCTSTR *params, LPVOID data);
```

де: hlog — дескриптор журналу;

etype — тип повідомлення (EVENTLOG_ERROR_TYPE — повідомлення про помилку, EVENTLOG_WARNING_TYPE — попередження, EVENTLOG_INFORMATION_TYPE — інформаційне повідомлення);

ecode — код повідомлення (відповідна константа із заголовного файла);

parnum – кількість параметрів повідомлення (елементів масиву params);

params — масив параметрів повідомлення (під час виведення повідомлення ці параметри будуть послідовно підставлені замість специфікацій підстановки).

Ця функція у разі помилки повертає значення FALSE:

```
const char *messages[] = { argv[0] };  // ім'я виконуваного файла
// HANDLE hlog = RegisterEventSource(...);
ReportEvent(hlog, EVENTLOG_INFORMATION_TYPE, 0,
    MSG_MYMSG, NULL, 1, 0, messages, NULL);
```

Унаслідок виконання цього фрагмента коду в журналі подій для джерела myapp у системі із встановленими українськими локальними налаштуваннями відобразиться таке повідомлення:

```
Програму запущено (виконується файл: c:\path\myapp.exe)
```

Для використання функцій роботи із журналом під час компонування застосування необхідно зазначати бібліотечний файл advapi32.lib.

18.7. Локальна безпека даних

Цей розділ присвячено організації локальної безпеки даних в операційних системах. Вона пов'язана із забезпеченням конфіденційності інформації в рамках локального комп'ютера. Як приклад реалізації технології буде наведено автоматичне шифрування даних на файлових системах.

Зазначимо, що є й інші підходи до реалізації локальної безпеки системи. Більшість сучасних ОС реалізують бібліотеки, що дають змогу програмістові шифрувати дані та виконувати інші операції, пов'язані з їхнім криптографічним перетворенням, безпосередньо у прикладних програмах. У системах лінії Windows XP для цього призначено інтерфейси CryptoAPI і SSPI [56], в UNIX-системах можна використати системний виклик `crypt()`.

18.7.1. Принципи шифрування даних на файлових системах

Механізми керування доступом до файлів не можуть запобігти несанкціонованому доступу до інформації у разі фізичного викрадення жорсткого диска. Зробивши це, зловмисник може підключити диск до комп'ютера із сумісною версією операційної системи, у якій він є привілейованим користувачем. Після реєстрації із правами такого користувача можна отримати доступ до всіх файлів на викраденому диску незалежно від того, які для них задані списки контролю доступу. Така проблема характерна для переносних комп'ютерів (ноутбуків), оскільки вони частіше потрапляють у чужі руки.

Щоб запобігти такому розвитку подій, необхідно організувати конфіденційне зберігання найціннішої інформації. Найчастіше це реалізують за допомогою шифрування даних на файловій системі.

Таке шифрування звичайно здійснюють на рівні драйвера файлової системи, який перехоплює спроби доступу до файла і шифрує та дешифрує його «на льоту» у разі, коли користувач надав необхідні дані (наприклад, ключ) для виконання цих операцій.

Далі в цьому розділі йтиметься про особливості підтримки таких файлових систем у Linux і Windows XP.

18.7.2. Підтримка шифрувальних файлових систем у Linux

У Linux є кілька реалізацій файлових систем із підтримкою шифрування даних. Найвідоміші з них CFS і TCFS.

Підтримка файлової системи CFS реалізована в режимі користувача і дає змогу шифрувати дані на будь-якій наявній файловій системі ціною певної втрати продуктивності. Як алгоритм шифрування використовують потрійний DES. Шифрування може бути застосоване для окремих каталогів і файлів.

Підтримка системи TCFS реалізована в режимі ядра, що дає змогу досягти вищої продуктивності і ступеня захисту. Встановлення підтримки цієї файлової системи, однак, складніше (потрібні внесення змін у код ядра Linux і його пере-

компіляція). Ця файлова система реалізує концепцію динамічних модулів шифрування, надаючи користувачу можливість вибору алгоритму і режиму шифрування, які будуть використані для конкретної файлової системи, окремого каталогу або файла.

18.7.3. Шифрувальна файлова система Windows XP

Засоби підтримки шифрування файлів у ОС лінії Windows XP описані під загальною назвою *шифрувальної файлової системи* (Encrypting File System, EFS). Для цього використовують драйвер файлової системи, розташований над драйвером NTFS.

Принципи роботи EFS

Реалізація EFS – це гібридна криптосистема із кількома рівнями шифрування.

- Безпосередньо для шифрування даних файла використовують симетричний алгоритм (посилений аналог DES, можливе використання інших алгоритмів). Ключ для нього (*ключ шифрування файла* – File Encryption Key, FEK) генерують випадково під час кожної спроби зашифрувати файл.
- Для шифрування FEK використовують алгоритм із відкритим ключем (RSA). Для кожного користувача генерують пару RSA-ключів, при цьому FEK шифрують відкритим ключем цієї пари. Результат шифрування FEK зберігають у заголовку файла. Крім того, передбачена можливість дешифрування файла довіреною особою (*агентом відновлення*, recovery agent) у разі втрати ключа користувачем. Для цього результат шифрування FEK відкритими ключами довірених агентів також зберігають у заголовку файла.
- Відкритий ключ користувача зберігають у вигляді сертифіката у *сховищі сертифікатів* (certificate store), розташованому в домашньому каталозі користувача на локальному комп'ютері. Крім того, у цьому сховищі містяться сертифікати всіх агентів відновлення.
- Для шифрування закритого ключа користувача використовують симетричний алгоритм RC4 із ключем, який система генерує випадково і періодично обновлює. Цей ключ називають *майстер-ключем* (master key). Результат шифрування закритого ключа майстер-ключем зберігають на файловій системі в домашньому каталозі користувача.
- Для шифрування майстер-ключа теж використовують симетричний алгоритм RC4, але із більшою довжиною ключа. Його генерують на основі застосування односторонньої хеш-функції SHA-1 до даних облікового запису користувача (його SID і паролю). Результат шифрування майстер-ключа цим ключем також зберігають на файловій системі.

Програмний інтерфейс EFS

Для шифрування файла або каталогу використовують функцію `EncryptFile()`, а для дешифрування – `DecryptFile()` [70]:

```
EncryptFile("myfile.txt");
DecryptFile("myfile.txt", 0);
```

Для перевірки того, чи файл зашифрований, використовують функцію FileEncryptionStatus():

```
BOOL FileEncryptionStatus(LPCTSTR fname, LPDWORD status);
```

де: fname — ім'я файла або каталогу;

status — покажчик на змінну, у яку заноситься інформація про підтримку шифрування для файла (FILE_ENCRYPTABLE — файл може бути зашифрований, FILE_IS_ENCRYPTED — файл зашифрований; інші значення показують, що шифрування не підтримується).

Ця функція повертає FALSE, якщо під час перевірки виникла помилка.

```
DWORD status;
if (FileEncryptionStatus("myfile.txt", &status)) {
    if (status == FILE_IS_ENCRYPTED) printf ("Файл зашифрований\n");
}
```

Для деяких каталогів має сенс заборонити шифрування. Для цього необхідно помістити в такий каталог файл desktop.ini з інформацією:

```
[Encryption]
Disable=1
```

18.8. Мережна безпека даних

У цьому розділі йтиметься про організацію мережної безпеки даних в операційних системах, яка пов'язана із забезпеченням конфіденційності інформації у разі її передавання мережею [10, 40]. Як приклад буде розглянуто засоби забезпечення мережної безпеки на різних рівнях мережної архітектури TCP/IP.

18.8.1. Шифрування каналів зв'язку

Захист даних у протоколах стека TCP/IP

Протоколи стека TCP/IP самі не мають достатнього рівня захисту. Фактично, основною їхньою метою була реалізація передавання даних між хостами і застосуваннями. Передбачалося, що для підтримки таких можливостей, як аутентифікація сторін або шифрування даних під час їхнього передавання через канал, ці протоколи потрібно розширити.

У цьому пункті покажемо, які розширення цих протоколів можуть бути запропоновані для реалізації безпеки даних.

Види шифрування каналів зв'язку

Різні рівні шифрування каналів зв'язку в рамках стека протоколів TCP/IP наведено нижче.

1. На канальному рівні *шифрують канали зв'язку* (link-by-link encryption). Для цього пакети автоматично шифрують під час передавання через незахищений фізичний канал і дешифрують після отримання. Ці дії звичайно виконує спеціалізоване апаратне забезпечення, у сучасних умовах таке шифрування обмежене модемними або виділеними лініями.

2. На мережному рівні виконують *наскрізне шифрування* (end-to-end encryption), при цьому пакети мережного рівня (наприклад, IP-дейтаграми) шифрують на хості-джерелі, залишають зашифрованими впродовж усього маршруту і автоматично дешифрують хостом-одержувачем. Передавання таких пакетів мережею не змінюється, тому що їхні заголовки залишаються незашифрованими. Прикладом реалізації шифрування на мережному рівні є архітектура протоколів IPSec.
3. На межі між транспортним і прикладним рівнями теж можна шифрувати дані. Наприклад, застосування може бути модифіковане таким чином, що у разі реалізації прикладного протоколу в ньому використовуватиметься не інтерфейс сокетів, а інтерфейс альтернативної бібліотеки, яка реалізує шифрування даних. Такий підхід застосовують у протоколі SSL/TLS.

Далі в цьому розділі буде розглянуто особливості реалізації двох підходів до шифрування каналів зв'язку: наскрізного шифрування на мережному та шифрування на транспортному рівнях.

18.8.2. Захист інформації на мережному рівні

Стандартним підходом для реалізації захисту інформації на мережному рівні стека протоколів TCP/IP є архітектура IPSec. Це набір протоколів, призначених для забезпечення наскрізного захисту пакетів, які передають між двома хостами.

Розглянемо мережну взаємодію у разі використання IPSec.

1. Хост Аліси генерує IP-дейтаграми для передавання мережею Бобові. Ці дейтаграми порівнюють із фільтрами IPSec, при цьому перевіряють, чи потрібно забезпечувати їхній захист. Фільтри IPSec задають у рамках політики безпеки IPSec, визначеної для хосту Аліси.
2. Якщо дейтаграма відповідає умові фільтра, хост Аліси починає взаємодіяти з хостом Боба із використанням протоколу *обміну ключами Інтернету* (Internet Key Exchange, IKE). При цьому відбувається взаємна аутентифікація сторін (на підставі сертифікатів або пароля), узгодження протоколу захисту пакетів і обмін ключами для шифрування інформації відповідно до протоколу роботи гібридної криптосистеми.
3. Програмне забезпечення хоста Аліси перетворює пакети відповідно до погодженого протоколу захисту пакетів і відсилає їх хосту Боба. У рамках IPSec є два протоколи захисту пакетів:
   - *заголовка аутентифікації* (Authentication Header, AH), при використанні якого пакети супроводжують одностороннім хешем для забезпечення цілісності, але не шифрують;
   - *інкапсулюючого захищеного навантаження* (Encapsulating Security Payload, ESP), який забезпечує наскрізне шифрування пакетів.

   Перетворені пакети — це звичайні IP-дейтаграми, їхнє пересилання мережею відбувається стандартним способом. Додаткові дані, необхідні для протоколів AH і ESP, інкапсулюють усередині дейтаграм.

4. Програмне забезпечення комп'ютера Боба перевіряє пакети на цілісність і дешифрує їхній вміст (для протоколу ESP). Далі IP-пакети передають звичайним способом на верхні рівні реалізації стека протоколів (транспортний і прикладний).

Оскільки протоколи IPSec визначені на мережному рівні, то у разі переходу на IPSec не потрібно змінювати наявне програмне забезпечення – усі дані, передані за допомогою TCP або UDP, будуть автоматично захищені.

18.8.3. Захист інформації на транспортному рівні

Протокол SSL/TLS

Найвідомішим протоколом захисту інформації на транспортному рівні був *протокол захищених сокетів* (Secure Socket Layer, SSL). Остання його версія (SSL 3.0) із невеликими змінами була прийнята як стандартний *протокол безпеки транспортного рівня* (Transport Layer Security, TLS 1.0); цей протокол називатимемо *SSL/TLS*.

Його розташовують між протоколом транспортного рівня (TCP) і протоколами прикладного рівня (наприклад, HTTP), а реалізацію зазвичай постачають у вигляді бібліотеки користувача, інтерфейс якої можна використати у застосуваннях замість інтерфейсу сокетів. Найвідомішою реалізацією бібліотеки підтримки цього протоколу є OpenSSL.

Застосування може реалізовувати протокол прикладного рівня поверх інтерфейсу SSL/TLS. Є кілька реалізацій стандартних протоколів прикладного рівня, які базуються на SSL/TLS, серед них реалізація протоколу HTTP (HTTPS), підтримувана більшістю сучасних веб-серверів і веб-браузерів.

Розглянемо послідовність кроків у разі використання SSL/TLS на прикладі HTTPS. Цей протокол ґрунтується на протоколі роботи гібридної криптосистеми.

1. Клієнт (веб-браузер) зв'язується із сервером із використанням протоколу HTTPS (наприклад, запросивши веб-документ, URL якого починається із https://). При цьому встановлюється TCP-з'єднання із портом 443 (стандартний порт HTTPS).
2. Клієнт і сервер погоджують алгоритми шифрування, які використовуватимуться під час обміну даними.
3. Сервер відсилає клієнтові свій відкритий ключ у складі сертифіката.
4. Клієнт верифікує сертифікат сервера за допомогою відкритого ключа довіреної сторони (центру сертифікації), що видала цей сертифікат. Сучасні веб-браузери постачають із ключами деяких центрів сертифікації, можна також встановлювати додаткові ключі. Якщо сертифікат вірний, аутентифікацію сервера вважають успішною.
5. Клієнт генерує сесійний ключ, шифрує його відкритим ключем сервера, шифрує HTTP-запит сесійним ключем і відсилає всю цю інформацію серверу.
6. Сервер розшифровує сесійний ключ своїм закритим ключем, HTTP-запит – сесійним ключем, формує HTTP-відповідь, шифрує її сесійним ключем, доповнює одностороннім хешем і відсилає клієнтові.
7. Клієнт дешифрує HTTP-відповідь сесійним ключем і перевіряє цілісність (для чого обчислює її односторонній хеш і порівнює із хешем, отриманим від сервера). Якщо цілісність дотримана, документ відобразиться.

Протокол SSH

Ще одним важливим протоколом безпеки транспортного рівня є протокол *безпечного командного інтерпретатора* (Secure Shell, *SSH*). Він багато в чому подібний до SSL/TLS, але використовується інакше. Найчастіше він застосовується для реалізації захищеного віддаленого доступу до системи.

Після встановлення з'єднання між SSH-клієнтом і SSH-сервером (коли завершена аутентифікація сторін і обмін ключами) SSH-сервер на віддаленому комп'ютері запускає копію командного інтерпретатора, яка використовує псевдотермінал. SSH-клієнт може взаємодіяти із цим інтерпретатором, емулюючи термінал.

Взаємодія користувача із системою фактично відбувається аналогічно до протоколу telnet, але при цьому весь канал зв'язку шифрують. Сьогодні у багатьох UNIX-системах використання SSH є обов'язковим (протокол telnet блокують із міркувань безпеки).

18.9. Атаки і боротьба з ними

У цьому розділі ознайомимося із деякими підходами, які можна використати для атаки на систему безпеки ОС. Через брак місця виклад буде обмежено атаками переповнення буфера і найпростішими прикладами відмови в обслуговуванні. Як технології запобігання атакам буде розглянуто організацію квот на ресурси і зміну кореневого каталогу застосування.

18.9.1. Переповнення буфера

Розповсюдженим типом атак на програмний код у сучасних ОС є *атаки переповнення буфера* (buffer overflow attacks) [44]. Усі вони використовують некоректний програмний код (переважно мовою С), що не перевіряє довжину буфера, у який записують зовнішні дані, отримані від користувача. Ось приклад такого коду:

```
#include <stdio.h>
void f() {
    char buf[128];
    gets(buf); // небезпечне одержання рядка даних зі стандартного вводу
}
```

Функція gets(), що входить у стандартну бібліотеку мови С, вводить рядок символів довільної довжини зі стандартного вводу і розміщує їх у буфері buf. При цьому сама функція не перевіряє, скільки символів насправді було введено і чи достатньо для них місця в буфері. У ситуації, коли користувач ввів більше ніж 128 символів, програма запише дані у пам'ять, розташовану за buf.

Для того щоб зрозуміти, у чому тут полягає небезпека, розглянемо організацію пам'яті, у якій виконують застосування (рис. 18.3).

Як бачимо, адреса повернення функції і локальні змінні (зокрема і буфер buf) розміщені у програмному стеку. Коли зловмисник введе значно більше символів, ніж може бути розміщено у buf, вони переповнять буфер і будуть записані поверх адреси повернення функції. У результаті після повернення із функції відбудеться

перехід за адресою, взятою із введеного зловмисником рядка. При цьому вміст рядка може бути ретельно підібраний так, щоб цей його фрагмент містив адресу коду, який бажає виконати зловмисник. Згаданий код може перебувати в іншій частині цього самого рядка і, наприклад, запускати командний інтерпретатор. Якщо застосування виконувалося із правами суперкористувача, запущений інтерпретатор дасть зловмисникові повний контроль над системою.

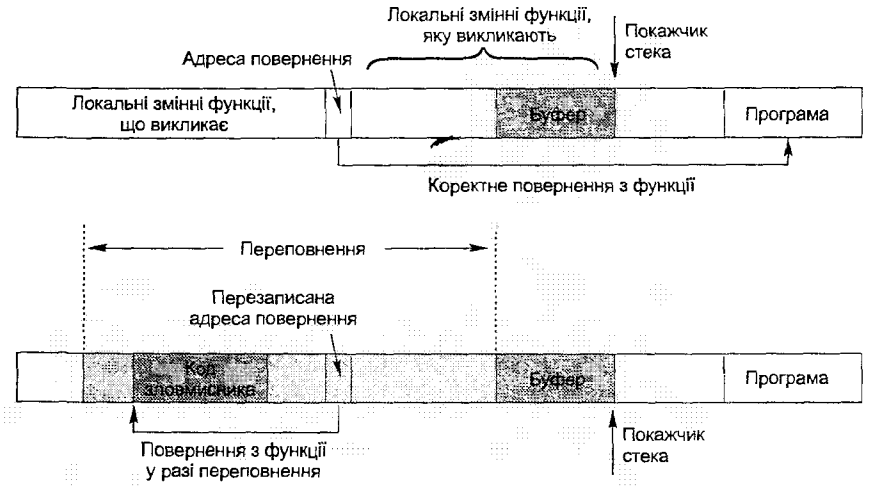

Рис. 18.3. Переповнення буфера

Для захисту від таких атак насамперед необхідно підвищувати якість розробки програмного забезпечення. Необхідно повністю відмовитися від використання функцій, які не перевіряють обсягу введених даних, замінивши їх варіантами, що роблять таку перевірку. Наприклад, замість функції `gets()` потрібно використати `fgets()`:

```
char buf[128];
fgets(buf, sizeof(buf), stdin);  // введення не більше 128 символів
```

Крім `gets()`, подібні проблеми виникають у разі використання таких функцій, як `strcpy()`, `strcat()` і `sprintf()`. Замість них потрібно використовувати відповідно `strncpy()`, `strncat()` і `snprintf()`.

Захист від переповнення буфера на рівні ОС може полягати в цілковитій забороні виконання коду, що перебуває у програмному стеку. Для Linux є виправлення до коду ядра, що реалізують це обмеження.

18.9.2. Відмова від обслуговування

Найпростіший приклад атаки відмови в обслуговуванні для UNIX-систем — так звана «fork-бомба»:

```
void main() {
    for (; ;) fork();
}
```

Очевидно, що процес, завантажений у пам'ять внаслідок запуску такої «бомби», почне негайно створювати свої власні копії, те саме продовжать робити ці копії і т. д. У старих версіях UNIX це могло швидко привести систему до нероботоздатного стану, і навіть сьогодні запуск такого застосування у системі із малим обсягом ресурсів здатний значно понизити її продуктивність.

Для боротьби із такими атаками необхідно встановлювати ліміти на ресурси. Зокрема, потрібно, щоб у системі було встановлено ліміт на максимальну кількість процесів, які можуть бути запущені під обліковим записом користувача. За замовчуванням ця кількість дорівнює 256, змінити її можна за допомогою системного виклику `setrlimit()`:

```
#include <sys/resource.h>
#include <unistd.h>
struct rlimit plimit;
plimit.rlim_max = 100;
setrlimit(RLIMIT_NPROC, &plimit);
```

Для ліквідації наслідків такої атаки не можна намагатися негайно завершити всі процеси-бомби, виконавши, наприклад, команду вилучення всіх процесів із заданим ім'ям:

```
# killall -KILL bomb
```

Річ у тому, що після запуску «бомби» у системі швидко створюється стільки процесів, що ліміт на їхню кількість виявиться вичерпаним. Після цього всі процеси-бомби перестають створювати нащадків і залишаються у нескінченному циклі. Як тільки після виконання `killall` завершиться один із таких процесів (а вони всі не можуть завершитись одночасно), загальна кількість процесів стане менша за ліміт. У результаті якийсь інший процес набору відразу отримує можливість виконати `fork()` і встигає це зробити до свого завершення. Фактично після знищення поточного набору процесів його місце негайно займає новий.

Правильним підходом буде спочатку призупинити всі процеси-бомби, а потім послати їм сигнал завершення:

```
# killall -STOP bomb
# killall -KILL bomb
```

Системний виклик `setrlimit()` може також бути використаний для встановлення інших квот на ресурси, зокрема обмежень на кількість відкритих файлів (першим параметром потрібно задати `RLIMIT_NOFILE`), на частку процесорного часу, яку використовує процес (`RLIMIT_CPU`), на розмір процесу у пам'яті (`RLIMIT_AS`).

Другим важливим засобом обмеження споживання ресурсів є квоти дискового простору.

18.9.3. Квоти дискового простору

Дискову квоту — встановлюване адміністратором обмеження на використання різних ресурсів файлової системи — задають для кожного користувача. Зазвичай до таких ресурсів належать кількість виділених кластерів або кількість створюваних файлів. Операційна система гарантує, що користувачі не зможуть перевищити виділені їм квоти.

Квоти звичайно реалізовують так. На диску організовують спеціальний файл (файл квот), що містить таблицю квот, куди заносять інформацію про квоти різних користувачів і те, які поточні обсяги ресурсів ними витрачені. Якщо користувач відкрив хоча б один файл, відповідний цьому користувачу елемент таблиці квот зчитують у пам'ять.

Файловий дескриптор, відкритий процесом, містить покажчик на елемент таблиці квот, що відповідає користувачу, який створив процес. У разі зміни розміру якогось із відкритих файлів інформацію про нову сумарну витрату ресурсу зберігають в елементі таблиці у пам'яті. Після того як користувач закриє всі свої файли, елемент таблиці квот зберігають на диску у файлі квот.

В елементі таблиці квот міститься така інформація: гнучкий і жорсткий ліміт на ресурс; поточний стан витрати ресурсу; максимально допустима і поточна кількість попереджень про перевищення гнучкого ліміту.

Спроба перевищити жорсткий ліміт завжди спричиняє помилку. Гнучкий ліміт може бути перевищений користувачем, але він при цьому отримує попередження (для кожного користувача визначено скінчений запас таких попереджень). Якщо користувач витратить усі попередження про перевищення гнучкого ліміту, його більше не допускають у систему без дозволу адміністратора.

Дискові квоти підтримують у Linux і в лінії Windows XP (для файлової системи NTFS).

18.9.4. Зміна кореневого каталогу застосування

Одним із ефективних способів запобігання або ослаблення дії різних атак в UNIX-системах є обмеження видимості файлової системи для застосувань. У такому разі кореневим каталогом ("/") для процесу стає не справжній кореневий каталог файлової системи, а один із його підкаталогів. Поза даним каталогом файлова система для цього процесу буде недоступною. Таку зміну відображення файлової системи здійснюють за допомогою системного виклику `chroot()`, у даному випадку кажуть про застосування зі зміненим кореневим каталогом або chroot-застосування.

Застосування зі зміненим кореневим каталогом не має доступу до більшості каталогів системи, тому заподіяна шкода від зловмисника, який отримав контроль над системою за допомогою такого застосування (наприклад, за допомогою атаки переповненням буфера), буде значно обмежена.

Виклик `chroot()` може бути виконаний тільки процесом із правами суперкористувача. Наведемо приклад його використання для того щоб зробити кореневим каталогом поточного процесу /home/user/newroot:

```
#include <unistd.h>
chroot("/home/user/newroot");
```

Зміну кореневого каталогу часто використовують у різних серверах. Так, деякі сучасні версії стандартних мережних серверів (сервер імен bind, поштовий сервер sendmail) як один із режимів підтримують роботу за умов зміненого кореневого каталогу.

Розглянемо послідовність завантаження сервера у разі використання `chroot()`.

1. Здійснюють підготовку до обслуговування клієнтів (завантажують необхідні динамічні бібліотеки і конфігураційні файли, відкривають файли журналу).
2. Виконують виклик `chroot()`.
3. Сервер починає очікування з'єднань від клієнтів та їхнє обслуговування. При цьому рекомендовано перейти в режим виконання із правами звичайного користувача за допомогою системного виклику `setuid()`.

Висновки

- Основними завданнями забезпечення безпеки комп'ютерних систем є аутентифікація, керування доступом, аудит, забезпечення конфіденційності, доступності та цілісності даних.
- Сучасні засоби підтримки безпеки ґрунтуються на таких криптографічних технологіях, як алгоритми із секретним і відкритим ключами, гібридні криптосистеми, цифрові підписи і сертифікати.
- Аутентифікація дає змогу впевнитися, що користувач є тим, за кого себе видає, і може бути допущений у систему. Для підтвердження особи користувача звичайно використовують пароль. Сучасні технології аутентифікації дають змогу уникнути передавання пароля каналом зв'язку у незашифрованому вигляді.
- Керування доступом (авторизація) дає змогу визначити дії, які авторизований користувач може виконувати із різними об'єктами у системі. Авторизація реалізована на основі визначення списків контролю доступу, пов'язаних із об'єктами у системі, а також списків можливостей, пов'язаних із окремими користувачами.
- Організація аудиту дає змогу зберігати інформацію про різні події у системі для подальшого аналізу. Інформацію зберігають у спеціальному журналі аудиту (системному журналі, журналі подій), вибір подій для реєстрації в цьому журналі визначає політика аудиту.
- Для забезпечення конфіденційності даних у рамках локальної системи використовують шифрувальні файлові системи або криптографічні API. Мережна безпека даних забезпечена їхнім шифруванням на різних рівнях мережної архітектури. Найчастіше таке шифрування виконують на мережному рівні або між транспортним і прикладним рівнями.

Контрольні запитання та завдання

1. Як можна реалізувати підтримку цілісності повідомлень і конфіденційності передачі даних у системі електронної пошти?
2. Наведено описи трьох систем керування доступом ОС:
   а) кожен файл містить список коректних паролів; користувач зобов'язаний пред'явити один із цих паролів, щоб відкрити файл;
   б) кожному файлу присвоюють ім'я з 256 випадкових символів, єдиний спосіб одержати доступ до файла — це задати його ім'я;

в) для кожного файла адміністратор системи задає список користувачів, яким заборонено доступ до файла.

Яку технологію керування доступом використовують у кожній з цих систем?

3. Поясніть, яким чином введення підтримки списків контролю доступу може розширити функціональність системи, яка базується на можливостях.
4. Що спільного й у чому основні відмінності між роботою з журналом файлової системи і використанням журналу безпеки?
5. Модифікуйте фонові застосування, розроблені під час виконання завдань 9 і 10 з розділу 17, так, щоб виведення діагностичної інформації виконувалося, відповідно, у системний журнал (Linux) і в журнал повідомлень (Windows XP).
6. Припустімо, що в системі зареєстровано 4000 користувачів, і необхідно, щоб 3990 з них могли одержати доступ до одного файла. Як досягти цього в UNIX? Які поліпшення схеми атрибутів безпеки UNIX можна запропонувати з урахуванням можливості виникнення такої ситуації?
7. Розробіть застосування для Windows XP, що створює каталог і встановлює права читання і записування у цей каталог для заданого користувача. Імена каталогу і користувача задають у командному рядку.
8. Як виконати атаку відмови в обслуговуванні з використанням підсистеми керування віртуальною пам'яттю сучасних ОС?
9. Мережні хробаки — це застосування, що автоматично поширюють свої копії доступними мережними з'єднаннями. Яким чином поява в системі такого хробака може призвести до атаки відмови в обслуговуванні, навіть якщо він не містить коду, що зумисно здійснює таку атаку?
10. Користувач `victim` залишив на якийсь час без догляду консоль Linux з відкритою сесією командного інтерпретатора. Користувач `attacker`, використовуючи цю консоль, виконав дії, що дають йому змогу надалі в будь-який момент одержати повний контроль над усіма даними `victim`. Незважаючи на те, що `victim` знає про дії `attacker`, перешкодити захопленню контролю над своїми даними він не може. Ні `victim`, ні `attacker` не мають права `root`. Які дії виконав `attacker`?
11. Яким чином користувач може дозволити іншим користувачам додавати рядки у файл, власником якого він є? Всі інші дії з файлом (вилучення, зміна рядків тощо) мають бути заборонені. Є як мінімум два принципово різних розв'язання цієї задачі. Розробіть їхній програмний код.

Розділ 19

Завантаження операційних систем

- ✦ Загальні принципи організації процесу завантаження
- ✦ Процес завантаження Linux і Windows XP

Завантаження операційної системи — складний процес, що стосується більшості підсистем ОС. Його повне розуміння неможливе без наявності базових знань про структуру різних компонентів системи.

19.1. Загальні принципи завантаження ОС

Тут зробимо короткий огляд загальних принципів організації завантаження операційних систем. Основну увагу буде приділено апаратній ініціалізації комп'ютера і принципам реалізації завантажувача ОС. Подальші етапи завантаження опишемо коротко, докладніше їх буде розглянуто на прикладах у розділах, присвячених завантаженню Linux і Windows XP.

19.1.1. Апаратна ініціалізація комп'ютера

Коли комп'ютер увімкнений в електромережу, він по суті порожній — усі його мікросхеми пам'яті містять випадкові значення, процесор не виконує код. Для початку процедури завантаження на процесор подають команду RESET (скидання). Після її прийняття, деякі регістри процесора (зокрема регістр лічильника команди) набувають фіксованих значень, і починається виконання коду за фізичною адресою 0xfffffff0. Апаратне забезпечення відображає цю адресу на спеціальну ділянку енергонезалежної пам'яті (ROM). Набір програм, що зберігається у ROM, за традицією називають BIOS (Basic Input/Output System, базова система введення/виведення), він включає набір керованих перериваннями низькорівневих процедур, які можна використати для керування пристроями, підключеними до комп'ютера.

Більшість сучасних ОС використовують BIOS тільки на етапі початкового завантаження (який називають bootstrapping). Після цього вони ніколи не звертаються до процедур BIOS і всі функції керування пристроями в ОС беруть на себе драйвери цих пристроїв. Річ у тому, що процедури BIOS можуть виконуватися тільки в реальному режимі процесора, а ядро — у захищеному режимі; крім того, звичайно код BIOS не має високої якості. Реальну адресацію використовують у коді BIOS тому, що тільки такі адреси виявляються доступними, коли комп'ютер тільки-но увімкнено.

Процедура початкового завантаження BIOS (bootstrap procedure) зводиться до чотирьох операцій.

1. Виконання набору тестів апаратного забезпечення для з'ясування, які пристрої в системі присутні та чи всі вони працюють коректно. Цей етап називають *самотестуванням після увімкнення живлення* (Power-On Self-Test, POST).
2. Ініціалізація апаратних пристроїв. Цей етап дуже важливий у сучасних архітектурах, заснованих на шині PCI, оскільки він гарантує, що всі пристрої працюватимуть без конфліктів у разі використання ліній переривань або портів введення-виведення. Наприкінці цього етапу буде відображено список установлених PCI-пристроїв.
3. Пошук і виконання початкового коду завантаження. Залежно від установок BIOS здійснюють спробу доступу (у заздалегідь визначеному порядку, який можна змінити) до першого сектора гнучкого диска, заданого жорсткого диска або компакт-диска. У жорсткому диску, як уже відомо з розділу 12, перший сектор називають головним завантажувальним записом (MBR).
4. Коли пристрій знайдено, BIOS копіює вміст його першого сектора в оперативну пам'ять (починаючи із фіксованої фізичної адреси 0x00007c00), виконує команду переходу на цю адресу і починає виконувати щойно завантажений код. За все інше відповідає операційна система.

19.1.2. Завантажувач ОС

Завантажувачем ОС (boot loader) називають програму, викликану кодом BIOS під час виконання процедури початкового завантаження для створення образу ядра операційної системи в оперативній пам'яті. Розглянемо основні принципи роботи найпростішого завантажувача в архітектурі PC.

Як зазначалося, BIOS починає виконувати код, який зберігається у MBR. Звичайно MBR містить таблицю розділів і невелику програму, що завантажує перший (завантажувальний) сектор одного із розділів у пам'ять і починає виконувати код, що перебуває в ньому, — зазвичай цей код називають кодом завантажувача ОС. Вибір розділу, з якого потрібно завантажити сектор, переважно здійснюють за допомогою прапорця активного розділу, заданого для елемента таблиці розділів. У такий спосіб можна завантажити тільки ту ОС, ядро якої перебуває на активному розділі, інші підходи розглянемо пізніше.

Завантажувач ОС звичайно записують у завантажувальний сектор під час інсталяції системи, тоді ж задається і код для MBR. Код найпростішого завантажувача зводиться до пошуку на диску ядра ОС (зазвичай це файл, що перебуває у фіксованому місці кореневого каталогу) і завантаження його у пам'ять.

19.1.3. Двоетапне завантаження

Використання завантажувального сектора для безпосереднього завантаження ядра ОС має такі недоліки:

- код завантажувача вимушено є дуже простим, тому в ньому не можливо виконувати складніші дії (наприклад, керувати завантаженням кількох ОС), більшість інших недоліків є наслідками цього;

- не можливо передавати параметри у завантажувач;
- процес обмежений описаною схемою перемикання із MBR на завантажувальний сектор, немає можливості керувати цим процесом;
- немає змоги завантажувати ядро з іншого розділу диска або із підкаталогу;
- ОС завжди буде запущена в реальному режимі процесора.

Для того щоб вирішити ці проблеми, код завантажувача ОС має бути ускладнений. Природно, що ускладнений код не поміститься в один сектор, тому запропоновано підхід, який отримав назву схеми двоетапного завантаження.

У цьому разі завантажувач розбивають на дві частини: завантажувач першого і другого етапів. Перший, як і раніше, зберігають у завантажувальному секторі диска (зазначимо, що під час використання цієї технології він може також зберігатися і у MBR), але тепер основним його завданням є пошук на диску і завантаження у пам'ять не ядра, а завантажувача другого етапу.

Завантажувач другого етапу — це повномасштабне застосування, яке отримує контроль над комп'ютером після виконання початкового завантаження (воно може бути виконане в реальному режимі процесора, а може перемикатися у привілейований режим). По суті, такий завантажувач є міні-ОС спеціалізованого призначення.

Розглянемо деякі можливості двоетапного завантажувача.

- У ньому можна керувати завантаженням кількох операційних систем. Особливо зручно це робити у завантажувачах, що приймають керування від MBR: при цьому завантажувач бере на себе пошук активного розділу і завантаження системи із нього. Конфігурацію такого завантажувача можна динамічно змінювати під час зміни розділів диска. Завантажувач може містити код доступу до різних файлових систем, код завантаження різних ядер тощо.
- Може надавати інтерфейс користувача, який зазвичай зводиться до відображення меню вибору завантажуваної операційної системи. Можливе також передавання введених користувачем параметрів у ядро перед його завантаженням.
- Його конфігурація може бути задана користувачем із завантаженої ОС. Для цього, наприклад, можна задати текстовий конфігураційний файл, зберегти його на диску і запустити спеціальну утиліту, що зробить синтаксичний розбір файла, перетворить його у внутрішнє відображення і збереже на диску у фіксованому місці, відомому завантажувачу.
- Не обмежений одним розділом і одним диском завантажувач може працювати з усіма дисками комп'ютера, завантажувати ядра, що перебувають у різних місцях на диску (зокрема всередині ієрархії каталогів).

Двоетапні завантажувачі надзвичайно поширені, їх можуть постачати разом з ОС (наприклад, lilo або GRUB для Linux, завантажувач Windows XP), а також вони можуть бути реалізовані як окремі продукти — менеджери завантаження (boot managers).

19.1.4. Завантаження та ініціалізація ядра

Код ядра зберігають в окремому файлі на диску, завантажувач повинен його знайти. Є різні підходи до реалізації завантаження ядра: воно може завантажитися

у пам'ять повністю або частинами (при цьому одні частини можуть завантажувати інші за потребою), або у частково стиснутому стані (а після виконання деяких попередніх операцій бути розпакованим).

У разі одноетапного завантаження ядро завантажують завжди в реальному режимі. У разі двоетапного — режим завантаження залежить від того, чи перемикають завантажувач другого етапу в захищений режим.

Після завантаження ядра у пам'ять керування передають за адресою спеціальної процедури, що починає процес ініціалізації ядра, і виконуються такі дії, як опитування та ініціалізація устаткування (зазвичай ініціалізують все устаткування — навіть те, що було вже проініціалізовано BIOS), ініціалізація підсистем ядра, завантаження та ініціалізація необхідних драйверів (насамперед диска та відеокарти), монтування кореневої файлової системи. Точна послідовність дій різна для різних ОС.

19.1.5. Завантаження компонентів системи

Після того як ядро завантажилося у пам'ять, починається завантаження різних компонентів системи. До них належать додаткові драйвери (які не були потрібні під час завантаження), системні фонові процеси тощо. Більшу частину цієї роботи виконують у режимі користувача. У результаті до моменту, коли користувач може почати працювати із системою, у пам'яті, крім ядра, присутній набір процесів, потрібних для повноцінної роботи.

Коли система завантажилася повністю, вона звичайно відображає запрошення для входу користувача (якщо вона з'єднана із терміналом), а коли користувач успішно ввійде у систему, для нього ініціалізують програмне забезпечення, що організовує його сесію (наприклад, запускають копію командного інтерпретатора).

19.2. Завантаження Linux

19.2.1. Особливості завантажувача Linux

Під час завантаження Linux використовують двоетапний завантажувач. Є кілька програмних продуктів, що реалізують такі завантажувачі, найвідоміший із них `lilo` (від linux loader). Він може бути встановлений як у MBR (замінивши там код, що завантажує перший сектор активного розділу), так і у завантажувальному секторі якогось (звичайно активного) розділу диска. Другий підхід є безпечнішим за умов, коли на комп'ютері встановлено кілька ОС у режимі мультизавантаження, оскільки деякі ОС можуть перезаписувати MBR за своєю ініціативою.

Перша частина `lilo`, записана у завантажувальний сектор або MBR, під час свого виконання готує пам'ять і завантажує в неї другу частину. Та зчитує із диска двійкове відображення карти наявних на комп'ютері варіантів завантаження (різні ОС, різні установки Linux тощо) і пропонує користувачу вибрати один із них (за допомогою підказки «LILO boot:»). Зазначимо, що вихідну версію карти варіантів завантаження створює користувач (системний адміністратор) у вигляді звичайного текстового файла /etc/lilo.conf. Після кожної зміни карти необхідно обновлювати її відображення на диску, використовуване завантажувачем. Для цього виконують команду (із правами root):

```
# lilo
```

Після вибору користувачем одного із варіантів завантаження поведінка завантажувача залежить від характеру файлової системи для розділу. У разі вибору розділу з іншою ОС (наприклад, Windows) зчитують у пам'ять і виконують завантажувальний сектор цього розділу (тому за допомогою `lilo` можна завантажити будь-яку ОС), якщо ж вибрано розділ із Linux, у пам'ять завантажують ядро системи (його адреса на диску міститься в карті варіантів завантаження). Після завантаження у пам'яті з'являється стиснуте ядро системи і придатний для виконання код двох функцій завантаження: `setup()` і `startup_32()`. Код завантажувача переходить до виконання функції `setup()`, починаючи ініціалізацію ядра.

19.2.2. Ініціалізація ядра

Перший етап ініціалізації ядра відбувається у реальному режимі процесора. Переважно здійснюється ініціалізація апаратних пристроїв (Linux не довіряє цього BIOS). Функція `setup()` визначає фізичний обсяг пам'яті у системі, ініціалізує клавіатуру, відеокарту, контролер жорсткого диска, деякі інші пристрої, перевизначає таблицю переривань, переводить процесор у захищений режим і передає управління функції `startup_32()`, код якої також перебуває поза стиснутим ядром.

Функція `startup_32()` задає сегментні регістри і стек, розпаковує образ ядра і розташовує його у пам'яті. Далі виконують код розпакованого ядра, при цьому керування спочатку дістає функція, яку також називають `startup_32()`. Вона формує середовище виконання для першого потоку ядра Linux («процес 0»), створює його стек (із цього моменту вважають, що він є), вмикає підтримку сторінкової організації пам'яті, задає початкові (порожні) оброблювачі переривань, визначає модель процесора і переходить до виконання функції `start_kernel()`.

Функцію `start_kernel()` виконують у межах потоку ядра «процес 0», завершуючи ініціалізацію ядра. Вона доводить до робочого стану практично кожен компонент ядра, зокрема:

- ініціалізує таблиці сторінок і всі дескриптори сторінок;
- остаточно ініціалізує таблицю переривань;
- ініціалізує кусковий розподілювач пам'яті;
- встановлює системні дату і час;
- виконує код ініціалізації драйверів пристроїв;
- робить доступною кореневу файлову систему (де розташовані файли, необхідні для завантаження системи).

Крім того, за допомогою функції `kernel_thread()` створюють потік ініціалізації («процес 1»), що виконує код функції `init()`. Цей потік створює інші потоки ядра і виконує програму `/sbin/init`, перетворюючись у перший у системі процес користувача `init`. Зазначимо, що для коректного завантаження `init` повинна бути доступна коренева файлова система із найважливішими розділюваними бібліотеками (каталог **/lib** має бути на тому самому розділі, що і кореневий каталог **/**).

На перетворенні цього потоку в `init` ініціалізація ядра завершена, функція `start_kernel()` переходить у нескінченний цикл простою (idle loop), не займаючи ресурсів процесора. Подальша ініціалізація системи відбувається під час виконання `init`.

19.2.3. Виконання процесу init

Процес `init` є предком усіх інших процесів у системі. Після запуску він читає свій конфігураційний файл /etc/inittab і запускає процеси, визначені в ньому. Набір процесів, які запускаються, залежить від дистрибутива Linux. Приклад виконання `init` під час завантаження системи Red Hat Linux наведено нижче.

Файл /etc/inittab визначає кілька рівнів роботи (runlevels). Кожен із них – це особлива програмна конфігурація системи, у якій може існувати тільки певна група процесів. Рівень роботи визначає режим функціонування ОС (однокористувацький, багатокористувацький, перезавантаження тощо). Стандартними рівнями роботи є рівні з 0 по 6. Ось найважливіші з них:

- 0 – завершення роботи (shutdown);
- 1 – однокористувацький режим (single user mode) – у ньому не дозволене виконання деяких фонових процесів, доступ до системи через мережу тощо;
- 3 – стандартний багатокористувацький режим (звичайно цей рівень задають за замовчуванням);
- 6 – перезавантаження (reboot).

Для деяких рівнів задано символьні синоніми (наприклад, для рівня 1 синонімом є рівень S). У файлі /etc/inittab визначено різні командні файли (із програмами, написаними мовою командного інтерпретатора, далі їх називатимемо скриптами), які повинні виконуватися для різних рівнів виконання.

Синтаксис рядка /etc/inittab такий:

```
ідентифікатор:рівень_роботи:дія:програма
```

Перший скрипт, який запускає `init`, визначений у рядку /etc/inittab із дією, заданою ключовим словом `sysinit`. Точне його ім'я залежить від поставки Linux, у Red Hat це /etc/rc.d/rc.sysinit. Його називають також стартовим скриптом. Ось відповідний рядок /etc/inittab:

```
si::sysinit:/etc/rc.d/rc.sysinit
```

Стартовий скрипт налаштовує базові системні сервіси, зокрема:

- час від часу перевіряє диски на помилки командою `fsck`;
- завантажує модулі ядра для драйверів пристроїв, які не повинні бути завантажені раніше;
- ініціалізує ділянку підкачування командою `swapon`;
- монтує файлові системи відповідно до файла /etc/fstab.

У файлі /etc/fstab задано, яка файлова система має бути змонтована у який каталог кореневої файлової системи. Кожний рядок цього файла відповідає одній операції монтування і містить інформацію про тип файлової системи, розділ, де вона розташована, точку монтування тощо.

Крім стартового скрипта, у каталозі /etc/rc.d перебувають кілька інших скриптів:

- /etc/rc.d/rc – викликають у разі зміни рівня виконання; як параметр він приймає номер рівня, звичайно запускає всі скрипти, що відповідають рівню (такі скрипти розглянемо трохи пізніше);

- /etc/rc.d/rc.local – викликають останнім під час завантаження системи; він містить специфічні для конкретної машини команди (зазвичай не рекомендують поміщати такі команди у стартовий скрипт, оскільки у разі перевстановлення або відновлення системи той файл стирають, а rc.local – ні).

Запуск системних фонових процесів та ініціалізацію системних служб (наприклад, мережних інтерфейсів) виконують із використанням набору файлів і підкаталогів у каталозі /etc.

У каталозі /etc/rc.d/init.d зберігають набір індивідуальних стартових скриптів, відповідальних за керування різними фоновими процесами і службами. Наприклад, скрипт /etc/rc.d/init.d/httpd відповідає за керування веб-сервером Apache. Кожний стартовий скрипт має обробляти отримані як параметри ключові слова start і stop, запускаючи і зупиняючи відповідний процес.

```
# /etc/rc.d/init.d/httpd start
```

Стартові скрипти не запускаються безпосередньо процесом init під час завантаження системи. Для організації такого запуску в /etc/rc.d є набір каталогів (rc0.d ... rc6.d), кожен із них містить символічні зв'язки, що вказують на стартові скрипти. У разі переходу на певний рівень виконання init запускає скрипт /etc/rc.d/rc, який переглядає всі зв'язки каталогу, що відповідає рівню, і виконує дії відповідно до їх імен.

Кожен зв'язок має деяке ім'я у форматі K*nn*ім'я або S*nn*ім'я (де *nn* – цифри, наприклад S70httpd), яке характеризується такими особливостями.

- Якщо ім'я починається на S, то на цьому рівні служба має бути запущена (якщо вона не запущена, потрібно виконати відповідний стартовий скрипт із параметром start); на K – на цьому рівні служба не повинна бути запущена (якщо вона запущена, її потрібно зупинити, виконавши стартовий скрипт із параметром stop).
- Число nn задає номер послідовності, що визначає порядок запуску або зупинки служб у разі переходу на рівень. Що більший nn, то пізніше виконається скрипт; важливо, щоб до цього часу вже були запущені всі служби, від яких залежить ця. Наприклад, ініціалізацію мережі потрібно виконувати якомога раніше, тому зв'язок, що вказує на скрипт, при цьому може бути такий: S20network.
- Ім'я зв'язку завершують іменем стартового скрипта, на який він вказує.

Наприклад, коли init переходить на рівень виконання 3, усі зв'язки, що починаються на S у каталозі /etc/rc.d/rc3.d, буде запущено в порядку їхніх номерів, і для кожного запуску буде задано параметр start. Після виконання всіх скриптів вимоги до рівня виконання 3 задовольняться.

В /etc/inittab має бути заданий рівень виконання за замовчуванням, для чого потрібно включити у файл рядок із ключовим словом initdefault. Система завершить своє завантаження після переходу на цей рівень. Звичайно, це – рівень 3.

```
id:3:initdefault:
```

Після запуску всіх скриптів для переходу на рівень за замовчуванням init завжди запускає спеціальну програму getty, що відповідає за зв'язок із користувачем через консоль і термінали (звичайно створюють кілька таких процесів – по одному на кожну консоль). Є різні реалізації цієї програми, у Linux популярними є agetty і mingetty. Саме getty видає підказку «login:».

Рядок у /etc/inittab, що задає запуск версії getty, має такий вигляд (ключове слово respawn означає, що процес буде перезапущено, якщо він припиниться):

```
1:2345:respawn:/sbin/mingetty tty1
```

Після того як користувач ввів своє ім'я, getty викликає програму /bin/login, що запитує пароль (видавши підказку «password:»), перевіряє його та ініціалізує сесію користувача. У більшості випадків це зводиться до запуску для користувача копії командного інтерпретатора (звичайно bash) у його домашньому каталозі. У результаті користувач може розпочати роботу із системою.

Процес init залишається у пам'яті і після завантаження. Він відповідає за автоматичний перезапуск процесів (для цього потрібно прописати програму в /etc/inittab із дією respawn, як getty); init також стає предком для всіх процесів, чий безпосередній предок припинив свою роботу.

Як зазначалося, у разі перезавантаження або зупинки система також переходить на певні рівні виконання, і при цьому виконуються скрипти з /etc/rc.d/init.d через зв'язки в каталогах для цих рівнів (rc0.d для зупинки, rc6.d для перезавантаження; такі зв'язки починаються на K). Для того щоб розпочати перезавантаження або зупинити систему, використовують спеціальну програму /sbin/shutdown, доступну лише суперкористувачу root. Консоль Linux також підтримує організацію перезавантаження від клавіатури натисканням на Ctrl+Alt+Del.

19.3. Завантаження Windows XP

Завантаження Windows XP починають стандартним способом — із передавання керування коду завантажувального сектора активного розділу диска. Головне його завдання — визначити місцезнаходження файла ntldr у кореневому каталозі цього розділу, завантажити його в пам'ять і передати керування на його точку входу. Зазначимо, що код завантажувального сектора залежить від того, яка файлова система встановлена для цього розділу: для FAT виконують один варіант, для NTFS — інший.

Файл ntldr можна розглядати як завантажувач другого етапу. Він починає своє виконання у 16-бітному режимі процесора, передусім переводить процесор у захищений режим і вмикає підтримку сторінкової організації пам'яті, після цього зчитує з кореневого каталогу файл boot.ini і робить його синтаксичний розбір. Ось фрагмент файла boot.ini:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows XP"
C:\="Windows 98"
```

Після тегу [boot loader] задано варіант завантаження за замовчуванням і час, після закінчення якого система автоматично завантажуватиметься відповідно до цього варіанта, після [operating systems] — список можливих варіантів завантаження. Для кожного варіанта може бути задано одну із кількох адрес завантаження:

- розділ із кореневим каталогом WINDOWS (для завантаження Windows XP);

- літерне позначення тому, на якому перебуває інша ОС;
- ім'я файла із зазначенням тому.

У разі зазначення літерного імені розділу (як у прикладі) ntldr знаходить на диску файл bootsec.dos (у якому після встановлення Windows XP зберігають завантажувальний сектор DOS або Consumer Windows, якщо поверх нього записаний завантажувальний сектор Windows XP), перемикає процесор у реальний режим і починає виконувати код цього завантажувального сектора.

Якщо задано ім'я файла, ntldr завантажуватиме файл із таким іменем; отже, якщо у файлі зберегти завантажувальний сектор іншої ОС, наприклад, Linux, ntldr зможе завантажити і його, для цього варіант завантаження має такий вигляд:

```
C:\bootsec.lnx="Linux"
```

Далі наведемо випадок завантаження Windows XP. Зазначимо, що розділ з установкою Windows XP у boot.ini не зобов'язаний збігатися із розділом, з якого відбувається завантаження, — таких розділів може бути кілька.

Коли є один варіант завантаження, система відразу починає завантажуватися, коли їх більше — відображають меню завантаження. Після вибору варіанта із меню ntldr запускає програму ntdetect.com, що в реальному режимі визначає базову конфігурацію комп'ютера (подібно до того, як це робила функція setup() для Linux — жодна із сучасних систем не довіряє цей код BIOS). Зібрану інформацію зберігають у системі, пізніше вона буде збережена в реєстрі. Внизу екрана з'являється текстовий індикатор прогресу. У цій ситуації можна натиснути на F8 і перейти в меню додаткових можливостей завантаження (у безпечному режимі тощо).

Потім ntldr завантажує у пам'ять ntoskrnl.exe (що містить ядро і виконавчу підсистему Windows XP), bootvid.dll (відеодрайвер за замовчуванням, що відповідає за відображення інформації під час завантаження), hal.dll (рівень абстрагування від устаткування) та основні файли реєстру. Після цього він визначає із реєстру, які драйвери встановлені в режимі запуску під час завантаження (це, наприклад, драйвер жорсткого диска) і завантажує їх (без ініціалізації). Буде завантажено також драйвер кореневої файлової системи. На цьому роль ntldr у завантаженні завершується, і він викликає головну функцію в ntoskrnl.exe для продовження завантаження.

Ініціалізація ntoskrnl.exe складається із двох етапів: фаз 0 і 1. Багато підсистем виконавчої системи приймають параметр, який показує, у якій фазі ініціалізації зараз перебуває система.

Під час виконання фази 0 переривання заборонені, на екрані нічого не відображається. Основною метою цього етапу є підготовка початкових структур даних, необхідних для розширеної ініціалізації під час виконання фази 1. Зазначимо, що менеджер процесів на цьому етапі ініціалізується майже повністю, за його допомогою створюють початковий об'єкт-процес із назвою Idle, процес System і системний потік для виконання ініціалізації фази 1.

Після завершення фази 0 переривання дозволені, і починає виконуватися системний потік. Під час виконання фази 1 керування екраном здійснює відеодрайвер bootvid.dll, що відображає завантажувальний екран і графічний індикатор прогресу на ньому (цей індикатор змінюватиметься упродовж всієї фази 1). Відбувається остаточна ініціалізація різних підсистем виконавчої системи (менеджера

об'єктів, планувальника, служби безпеки, менеджера віртуальної пам'яті, менеджера кеша тощо). Під час ініціалізації підсистеми введення-виведення (яка займає до 50 % часу цієї фази) відбувається підготовка необхідних структур даних, ініціалізація драйверів із запуском під час завантаження (boot-start), завантаження та ініціалізація драйверів із системним запуском (system-start). Фаза 1 завершується запуском менеджера сесій (smss.exe).

Подальше завантаження виконують три системні процеси, розглянуті у розділі 2: менеджер сесій smss.exe, процес реєстрації у системі winlogon.exe і менеджер керування сервісами (SCM, services.exe). Основним завданням менеджера сесій є завантаження та ініціалізація всіх компонентів підсистеми Win32 (як режиму користувача, так і режиму ядра), а також остаточна ініціалізація реєстру і запуск winlogon.exe.

Процес реєстрації у системі запускає менеджер керування сервісами і менеджер аутентифікації, а також організовує реєстрацію користувачів у системі, як описано у пункті 18.5.2.

Менеджер сервісів (SCM) завантажує та ініціалізує сервіси режиму користувача, встановлені в режимі автоматичного завантаження. Цей процес може тривати вже після початку інтерактивної роботи користувачів. Після ініціалізації сервісів завантаження вважають успішним.

Висновки

- Жодна операційна система не може розпочати роботу без виконання процедури завантаження та ініціалізації. Під час цього процесу спочатку виконується найпростіший завантажувач ОС, що перебуває у фіксованому місці жорсткого диска, потім він відшукує ядро ОС і завантажує його у пам'ять. Ядро в свою чергу ініціалізує свої внутрішні структури та апаратне забезпечення і передає керування процесам користувача або системним процесам, які завершують процес ініціалізації.
- Широко розповсюджене двоетапне завантаження, під час якого завантажувач ОС передає керування складнішому завантажувачу другого етапу, який може керувати завантаженням кількох систем, установлених на комп'ютері. Такий завантажувач виконує основні дії щодо взаємодії із користувачем і завантаження потрібної системи.

Контрольні запитання та завдання

1. Які додаткові можливості адміністрування надають ОС, у яких є підтримка завантаження ядра системи з флоппі-диска?
2. Вкажіть переваги встановлення завантажувача системи в MBR порівняно з його встановленням у завантажувальний сектор одного з розділів диска.
3. Комп'ютерна система включає кілька НЖМД, на кожному з яких встановлена своя файлова система. Опишіть, у якій послідовності ці системи повинні бути змонтовані під час завантаження. Чи є необхідність у ході завантаження одержувати доступ до диска без використання засобів ядра?

4. Ідентифікатор процесу для init дорівнює одиниці, він менший, ніж у будь-якого потоку ядра Linux. У той же час init стає процесом пізніше, ніж будуть створені потоки ядра. Як можна це пояснити?
5. Альтернативним способом завантаження ОС є підхід, реалізований для Linux утилітою `loadlin`. Ця утиліта виконувалася під керуванням MS-DOS або Windows і завантажувала ядро Linux з файлової системи FAT. Після завантаження ядра в пам'ять подальший процес ішов, як описано в розділі 19.2. Назвіть переваги і недоліки такого підходу.

Розділ 20

Багатопроцесорні та розподілені системи

- Багатопроцесорні системи
- Базові технології розробки розподілених систем
- Організація віддаленого виклику процедур
- Синхронізація та координація розподілених застосувань
- Особливості реалізації розподілених файлових систем
- Сучасні архітектури розподілених обчислень
- Кластери і grid-системи

Сьогодні є два основні підходи до збільшення обчислювальної потужності комп'ютерних систем. Перший з них пов'язаний із підвищенням тактової частоти процесора. При цьому у розробників виникають технологічні проблеми, які пов'язані з необхідністю організовувати охолодження процесорів і тим, що швидкість поширення сигналів обмежена. Крім того, єдиний процесор системи є її «вузьким місцем» у надійності – його вихід із ладу призводить до неминучого краху всієї системи.

Альтернативним підходом, про який ітиметься в цьому розділі, є організація паралельних обчислень на кількох процесорах. З одного боку, внаслідок збільшення кількості процесорів можна досягти більшої потужності, ніж доступна на цей момент для однопроцесорних систем. З іншого, такі системи мають більшу стійкість до збоїв – у разі виходу одного із процесорів із ладу на його місці можна використати інший.

Як було зазначено в розділі 1, можна виділити дві основні категорії систем, що використовують кілька процесорів. У *багатопроцесорних системах* набір процесорів перебуває в одному корпусі та використовує спільну пам'ять (а також периферійні пристрої). У *розподілених системах* процесори перебувають у складі окремих комп'ютерів, з'єднаних мережею. Паралельні обчислення організовані на базі спеціального програмного забезпечення, що приховує наявність мережі від користувачів системи.

20.1. Багатопроцесорні системи

Тут зупинимося на основних типах багатопроцесорних систем і особливостях їхньої підтримки в сучасних ОС на прикладі Linux і Windows XP.

20.1.1. Типи багатопроцесорних систем

Залежно від особливостей апаратної реалізації, багатопроцесорні системи бувають такі:

- з *однорідним доступом до пам'яті* або *UMA-системи* (Uniform Memory Access), у яких доступ до будь-якої адреси у пам'яті здійснюється з однаковою швидкістю;
- *з неоднорідним доступом до пам'яті* або *NUMA-системи* (Non-Uniform Memory Access), для яких це не виконується.

Однорідний доступ до пам'яті

Архітектура UMA-систем передбачає, що доступ будь-якого процесора до будь-якого модуля пам'яті відбувається однаково. Найпоширенішим підходом до реалізації такої системи є *архітектура зі спільною шиною*, коли всі процесори і модулі пам'яті з'єднані між собою спільною шиною пам'яті. У разі необхідності отримати доступ до пам'яті процесор перевіряє, чи не зайнята шина, і, якщо вона вільна, виконує фізичний доступ за заданою адресою.

Якщо шина зайнята, процесор чекає, поки вона не звільниться. Необхідність такого очікування є головним недоліком базової архітектури зі спільною шиною. Кількість процесорів, які можна використати в цій архітектурі, невелика (від 2 до 4).

Найпоширенішим способом зменшення часу очікування на спільній шині є оснащення кожного процесора власним апаратним кешем досить великого обсягу. При цьому, якщо відбувається вдале звертання до кеша, доступ до шини не потрібен. Навантаження на шину знижується, отже, у системі може підтримуватися більша кількість процесорів.

У цьому разі, однак, виникає інша проблема, пов'язана із необхідністю підтримки *когерентності кеша* (cache coherence) — погодженості даних, які перебувають у кешах різних процесорів.

Роз'яснимо це поняття. Є ймовірність, що одна й та саме ділянка пам'яті (наприклад, яка відповідає деякій структурі даних) буде одночасно збережена в кешах кількох процесорів. Розглянемо ситуацію, коли код одного з потоків змінює цю структуру даних. У результаті зміниться вміст основної пам'яті та кеш процесора, на якому виконувався цей потік. Вміст кешів інших процесорів при цьому залишиться незмінним і перестане відповідати даним, які перебувають в основній пам'яті, — кеш втратить когерентність.

Забезпечення когерентності кеша спричиняє зниження продуктивності. Необхідно, аби процесор, що змінює дані в пам'яті, передавав шиною спеціальний сигнал, який сповіщає інші процесори про цю зміну. У разі отримання сигналу кожен процесор, який визначив, що він кешував ті ж самі дані, має вилучити їх із кеша (зробити перехресне очищення кеша, cross invalidation). Періодичні звертання процесорів до шини для виявлення сигналу і перехресне очищення кеша займають багато часу. Крім того, підвищується ймовірність промаху під час доступу до кеша.

Незважаючи на ці проблеми, UMA-архітектуру із когерентним кешем широко використовують на практиці.

Неоднорідний доступ до пам'яті

Реалізація когерентного кеша не вирішує всіх проблеми, пов'язаних із наявністю спільної шини в UMA-архітектурі. Для створення багатопроцесорних систем, які

розраховані на значну кількість процесорів (більше ніж 100) і матимуть можливість подальшого масштабування, потрібно використати неоднорідний доступ до пам'яті (NUMA-архітектуру).

В NUMA-архітектурі із кожним процесором пов'язують його власну локальну пам'ять. Єдиний для всіх процесорів адресний простір при цьому збережено — кожен процесор у системі може одержати доступ до локальної пам'яті будь-якого іншого процесора, але доступ до такої віддаленої пам'яті відбувається повільніше, ніж до локальної.

Фактично NUMA-система складається з набору вузлів (nodes), кожен із яких містить один або кілька процесорів та мікросхеми їхньої локальної пам'яті (а також, можливо, засоби введення-виведення). Вузли з'єднані між собою спільною шиною. Очевидно, що доступ до локальної пам'яті не вимагає звертання до шини, внаслідок чого навантаження на неї значно знижується. Крім того, звичайно організовують когерентний кеш для операцій доступу до цієї шини (системи із таким кешем називають CC-NUMA — Cache-Coherent NUMA).

20.1.2. Підтримка багатопроцесорності в операційних системах

Розглянемо підходи до реалізації підтримки багатопроцесорності в ОС на програмному рівні.

Асиметрична багатопроцесорність

У разі використання асиметричної багатопроцесорності кожен процесор виконує код операційної системи незалежно від інших процесорів. Кожна копія ОС може бути завантажена в окрему ділянку пам'яті, можливе також спільне використання коду ОС різними процесорами з виділенням окремих ділянок пам'яті для даних.

Цей підхід було використано на ранніх стадіях розвитку підтримки багатопроцесорних архітектур в ОС. Наведемо його недоліки.

- Усі процеси користувача деякої копії ОС виконуються на тому самому процесорі, що й сама копія. Немає можливості організувати паралельне виконання коду в рамках окремого процесу, не можна вирівнювати навантаження на окремі процесори і на пам'ять.
- Неможливо організувати дисковий кеш через те, що копії ОС різних процесорів кешуватимуть дискові блоки окремо. Якщо різні процесори одночасно модифікують один і той самий дисковий блок у кеші, а потім спробують зберегти ці зміни на диск, втратиться інформація, оскільки тільки одна з цих змін буде справді записана на диск.

Симетрична багатопроцесорність

Основним підходом, який застосовують нині для підтримки UMA-архітектур, є *симетрична багатопроцесорність* (SMP). У даному разі у спільну пам'ять завантажують єдину копію операційної системи і всіх її даних, при цьому її код може бути виконаний кожним із процесорів або кількома процесорами одночасно.

Особливості SMP-систем наведено нижче.

- Усі процесори системи доступні з коду ОС. Планувальник ОС може організувати виконання її коду або коду потоку користувача на будь-якому процесорі.
- Для всіх процесорів доступні спільні дані, при цьому когерентність кеша підтримується апаратно.
- Потоки користувача і потоки ядра можуть виконуватися паралельно на різних процесорах. Під час виконання потік може мігрувати із процесора на процесор.
- Спроба повторного читання одних і тих самих даних процесором CPU_A може дати інший результат внаслідок того, що ці дані були змінені процесором CPU_B.
- У системі можливе вирівнювання навантаження між процесорами, для чого планувальник ОС може передавати новий потік для виконання найменш завантаженому процесору.
- Додавання нового процесора у систему автоматично робить його доступним для виконання коду ОС або процесів користувача. При цьому навантаження на інші процесори автоматично знижується.

Для того щоб скористатися перевагами багатопроцесорної архітектури, код ОС має бути багатопотоковим і реентерабельним. При цьому необхідна підтримка синхронізації на рівні ядра.

Найпримітивнішим підходом до забезпечення синхронізації є велике блокування ядра (big kernel lock). При цьому кожен процесор перед виконанням будь-якого коду ОС займає глобальний м'ютекс. Цей підхід неефективний, оскільки в конкретний момент часу код ОС може бути виконаний тільки на одному процесорі.

Сучасні ОС реалізують гнучкіший підхід, у якому код ядра розбивають на незалежні критичні ділянки, із кожною з яких пов'язують окремий м'ютекс.

Підтримка NUMA-архітектур

ОС, що підтримує NUMA-архітектуру, має прагнути до підвищення продуктивності внаслідок планування потоків для виконання на процесорах, що перебувають у тих самих вузлах, що і пам'ять, використовувана цими потоками. Природно, що повністю уникнути звертань до віддаленої пам'яті неможливо, але їхня кількість має бути мінімальною.

Звичайно для організації більш якісного планування у системі підтримують структуру даних, що описує топологію конкретної NUMA-системи (вузли, їхні характеристики і зв'язки між ними). Крім того, мають бути передбачені системні виклики, що дають змогу задавати вузли, на яких виконуватиметься потік.

20.1.3. Продуктивність багатопроцесорних систем

Масштабування навантаження

Під *масштабуванням навантаження* (workload scalability) у SMP-системах розуміють вплив додавання нових процесорів на продуктивність системи. У реальних умовах воно залежить від багатьох факторів.

- У разі збільшення кількості процесорів зростає навантаження на системну шину та пам'ять і, як наслідок, ціна промаху кеша.

- Кількість промахів кеша при цьому теж збільшується внаслідок того, що в системі збільшено кількість потоків, які потрібно планувати.
- Що більше процесорів, то більше зусиль потрібно докладати для забезпечення когерентності кеша.
- Кількість блокувань у системі зростає із ростом кількості процесорів.

Найбільший рівень масштабування навантаження досягають для потоків, обмежених можливостями процесора, найменший — для потоків, обмежених можливостями пристроїв введення-виведення.

Продуктивність окремих застосувань

Розглянемо, яким чином впливає наявність кількох процесорів на час виконання програмного коду.

Багатопроцесорність дає змогу поліпшити характеристики програми тільки тоді, коли в ній наявний паралелізм (як було зазначено в розділі 3.2, за умов багатопроцесорності може бути реалізовано справжній паралелізм, коли окремі частини програми виконуються одночасно кількома процесорами). При цьому для того щоб ОС мала можливість організувати такий паралелізм, код програми має бути багатопотоковим. Якщо програма не використовує багатопотоковість, її виконання у багатопроцесорній системі може спричинити зниження продуктивності через очікування на додаткових блокуваннях і міграцію між процесорами.

Крім того, навіть якщо програма є багатопотоковою, максимальне поліпшення її продуктивності обмежене відповідно до закону Амдала

$$S = \frac{1}{\left(\frac{T_{\text{посл}}}{T} + \frac{(1 - T_{\text{посл}}/T)}{n}\right)},$$

де S — виграш у швидкості виконання; T — загальний обсяг коду, $T_{\text{посл}}$ — обсяг коду, що не може бути виконаний паралельно, n — кількість процесорів.

20.1.4. Планування у багатопроцесорних системах

Головною особливістю планування у багатопроцесорних системах є його двовимірність. Крім прийняття рішення про те, який потік потрібно поставити на виконання наступним, необхідно визначити, на якому процесорі він має виконуватися. Крім того, важливо виділяти взаємозалежні потоки, що їх доцільно виконувати паралельно на кількох процесорах, аби їм було простіше взаємодіяти один із одним. У цьому розділі розглянемо деякі підходи до організації планування, які враховують ці фактори, а у наступному — важливе поняття спорідненості процесора, що впливає на організацію планування у багатопроцесорних системах.

Планування з розподілом часу

Найпростішим способом організації багатопроцесорного планування незалежних потоків є використання структури даних для готових потоків, спільної для всіх процесорів. Прикладом такої структури може бути багаторівнева черга, яка використовується під час планування із пріоритетами.

Коли потік на одному з процесорів завершує роботу або призупиняється, цей процесор починає виконувати код планувальника ОС. Планувальник при цьому блокує чергу готових потоків, ставить на виконання потік із найвищим пріоритетом і вилучає його керуючий блок із черги. Наступний за пріоритетом потік почне виконуватися на наступному звільненому процесорі і т. д. Такий підхід називають плануванням із розподілом часу, оскільки, як і у традиційних системах із розподілом часу, щоразу приймають рішення щодо використання одного процесора і виконання одного потоку.

Головним недоліком цього підходу є високий ступінь паралелізму доступу до черги готових потоків, що може стати «вузьким місцем» системи. Є ймовірність того, що більшу частину часу потоки проводитимуть в очікуванні на м'ютексі, який захищає чергу. Крім того, немає можливості уникнути перемикання контексту в разі призупинення потоку і подальшої його міграції на інший процесор.

Планування з розподілом простору

Планування з розподілом часу не пристосоване до організації виконання потоків, пов'язаних між собою, оскільки кожен потік розглядають окремо. Для організації виконання пов'язаних потоків необхідно одночасно розглядати кілька процесорів і розподіляти по них набір потоків. Цей підхід називають плануванням із розподілом простору.

Найефективнішим алгоритмом планування із розподілом простору є *бригадне планування* (gang scheduling). Цей алгоритм працює так.

1. Пов'язані потоки (наприклад, потоки одного процесу) одночасно запускають на виконання на максимально можливій кількості процесорів. Такі потоки становлять бригаду (gang).
2. Усі потоки бригади виконуються впродовж однакового для всіх кванта часу.
3. Після вичерпання кванта часу відбувається повне перепланування для всіх процесорів. Виконання починають потоки іншої бригади.

Якщо кількість потоків бригади менша, ніж кількість процесорів, виконання можуть почати потоки кількох бригад, якщо більша — виконується частина потоків бригади, а інші будуть заплановані до виконання упродовж наступного кванта часу.

Робота цього алгоритму показана на рис. 20.1. По вертикалі відкладено моменти часу, по горизонталі — процесори. Буквами позначено процеси (бригади), цифрами індексу — номери потоків.

	CPU1	CPU2	CPU3	CPU4	CPU5
T1	A_1	A_2	A_3	A_4	A_5
T2	B_1	B_2	B_3	C_1	C_2
T3	D_1	D_2	D_3	D_4	E_1
T4	E_2	E_3	E_4	E_5	E_6
T5	A_1	A_2	A_3	A_4	A_5

Рис. 20.1. Бригадне планування

20.1.5. Спорідненість процесора

Під *спорідненістю процесора* (CPU affinity) розуміють імовірність того, що потік буде запланований для виконання на процесорі, що виконував код цього потоку

минулого разу [31, 79]. Висока спорідненість означає малу ймовірність міграції потоку між процесорами під час його виконання.

Типи спорідненості процесора

Є два типи спорідненості процесора – *м'яка* (soft affinity) і *жорстка* (hard affinity).

М'яка або натуральна спорідненість – властивість планувальника підтримувати виконання потоку на одному й тому самому процесорі упродовж максимального проміжку часу. М'яка спорідненість не є вимогою, обов'язковою для виконання, у разі необхідності потік може мігрувати на інший процесор, але це має траплятися якомога рідше.

Відсутність реалізації м'якої спорідненості у планувальнику ОС може спричинити появу ефекту пінг-понгу (ping-pong effect), коли кожне перемикання контексту спричиняє міграцію потоку на інший процесор.

Жорстку спорідненість задають для окремого потоку, це – явне обмеження набору процесорів, на яких йому дозволено виконуватися. У такому випадку кажуть, що потік прив'язаний до одного або кількох процесорів. Вимога жорсткої спорідненості є обов'язковою (наприклад, якщо визначено, що потік може бути виконаний тільки на процесорі 0, процесор 1 для нього буде недоступний). Для задання жорсткої спорідненості звичайно реалізують спеціальний системний виклик.

Переваги задання спорідненості процесора

Опишемо деякі переваги, які надає задання спорідненості процесора для потоків.

Найважливішою є підвищення ефективності використання апаратного кеша. Що рідше мігрують потоки між процесами, то нижча ймовірність очищення кеша. Це пов'язано з тим, що будь-яка спроба змінити дані в коді потоку, який почав виконання на новому процесорі, спричиняє перехресне очищення кеша для процесорів, де він виконувався раніше. Крім того, після міграції на новий процесор потік ніколи не знаходить у його кеші «свої» дані, внаслідок чого відсоток влучень кеша зменшується. Особливо продуктивність знижується внаслідок ефекту пінг-понгу.

Друга перевага полягає в тому, що, задавши жорстку спорідненість, можна виділяти потоки, яким для виконання потрібні гарантовані процесорні ресурси (наприклад, потоки реального часу). У цьому разі можна прив'язати до деякої підмножини процесорів усі потоки, крім одного, а виділений потік – до інших процесорів. Наприклад, якщо прив'язати конкретний потік до одного із процесорів системи, а інші потоки виконувати на всіх інших процесорах, весь процесор опиниться в розпорядженні виділеного потоку, що дасть змогу контролювати його виконання в реальному масштабі часу.

Крім того, жорстка спорідненість дає можливість ефективніше розподіляти системне навантаження процесорами. Наприклад, коли відомо, що у системі з чотирма процесорами постійно виконуються процеси A і B, при цьому виконання A вимагає 75 % процесорного часу, а B – 25 %, доцільно прив'язати процес B до одного із процесорів, а процес A – до решти трьох.

Маска спорідненості

Задання жорсткої спорідненості для потоку відбувається із використанням *маски спорідненості* (affinity mask) – бітової маски, кожний біт якої відповідає процесору, присутньому у системі. Коли такий біт увімкнуто, то це означає, що потоку

дозволено виконуватися на відповідному процесорі; коли ні, то це може означати, що потік не може бути виконаний на відповідному процесорі або процесор відсутній у системі. Цю маску передають як параметр у системний виклик задання жорсткої спорідненості, після встановлення її звичайно зберігають у керуючому блоці потоку. Далі буде наведено приклади прив'язання потоків до процесорів у Linux і Windows XP за допомогою задання маски спорідненості.

20.1.6. Підтримка багатопроцесорності в Linux

Багатопроцесорні архітектури в Linux підтримують, починаючи із ядра версії 2.0 (1996 рік). Підтримка багатопроцесорності реалізована в засобах синхронізації процесів і планувальнику процесів.

Основи синхронізації у ядрі Linux описано в пункті 5.4.1. Первісна реалізація підтримки SMP ґрунтувалася на великому блокуванні ядра. Сучасні версії ядра підтримують підхід із розбиванням коду на окремі критичні секції.

Багатопроцесорне планування

Ядро Linux версії 2.4 реалізовувало стандартне планування із розподілом часу. Повна реалізація планування, розрахована на використання у багатопроцесорних архітектурах, уперше з'явилася у ядрі версії 2.6 [83].

- Реалізовано підтримку м'якої спорідненості, засновану на тому, що для кожного процесора підтримують окрему чергу готових процесів. До виконання на процесорі допускають лише процеси із його черги. Періодично відбувається балансування всіх черг. Усе це зменшує ймовірність міграції процесів між процесорами і запобігає появі ефекту пінг-понгу.
- Реалізовано системні виклики, що дають змогу задавати жорстку спорідненість для процесів [79].

Для встановлення маски спорідненості використовують системний виклик sched_setaffinity():

```
#include <sched.h>
long sched_setaffinity(pid_t pid, unsigned int len,
    unsigned long *user_mask_ptr);
```

де: pid — ідентифікатор процесу (0 — поточний процес);

len — довжина маски;

user_mask_ptr — покажчик на змінну, що містить маску.

У разі помилки (наприклад, якщо задано маску, що містить тільки відсутні процесори) цей виклик повертає негативне значення. Зазначимо, що задання масок для процесів, запущених іншими користувачами, дозволено тільки для root.

```
unsigned long mask = 7; // прив'язати процес до процесорів 0, 1 і 2
sched_setaffinity(0, sizeof(mask), &mask);
```

Для того щоб дізнатися про поточне значення маски спорідненості, використовують системний виклик sched_getaffinity():

```
unsigned long mask;
sched_getaffinity(0, sizeof(mask), &mask);
printf("маска спорідненості: %08lx", mask);
```

Реалізація жорсткої спорідненості в ядрі

Маску процесу зберігають у його дескрипторі task_struct як поле cpus_allowed. У разі будь-якої спроби міграції процесу на новий процесор ядро перевіряє, чи увімкнуто в цьому полі біт, що відповідає тому процесору. Коли біт не встановлено, міграція процесу не відбувається. Крім того, якщо внаслідок зміни маски з'ясується, що процес виконується на недопустимому процесорі, він негайно мігрує на один із вказаних у новому значенні маски.

Підтримка NUMA-систем

Підтримка архітектури NUMA була вперше реалізована у ядрі версії 2.6. Основою такої підтримки є внутрішнє відображення топології вузлів, для роботи із якими призначений спеціальний *програмний інтерфейс топології* (Topology API). Інформацію про топологію вузлів використовують під час планування процесів. Основною метою планування є підвищення ймовірності використання локальної пам'яті.

20.1.7. Підтримка багатопроцесорності у Windows XP

За умов багатопроцесорної системи планувальник Windows XP визначає порядок виконання потоків і процесори, на яких вони мають виконуватися. При цьому за замовчуванням підтримують м'яку спорідненість. Крім того, аналогічно до Linux у системі може бути задана жорстка спорідненість на основі маски спорідненості, а серед процесорів, заданих у масці спорідненості потоку, додатково вибирають *ідеальний процесор* (ideal processor). Планувальник планує потік для виконання на ідеальному процесорі, якщо він є доступним у цей момент.

Маска спорідненості потоку, номер ідеального процесора і номер процесора, на якому потік виконувався останній раз, містяться в керуючому блоці потоку (KTHREAD), маска спорідненості процесу — у блоці KPROCESS. Під час створення потоку його маску спорідненості ініціалізують значенням маски спорідненості процесу.

У разі постановки потоку на виконання відбуваються такі дії.

1. Процесором, на якому виконуватиметься потік, планувальник намагається зробити ідеальний процесор.
2. Якщо ідеальний процесор недоступний, вибирають процесор, на якому потік виконувався востаннє.
3. Якщо і цей процесор зайнятий, вибирають інший вільний процесор або процесор, на якому виконується потік із нижчим пріоритетом.

Задання жорсткої спорідненості та ідеального процесора

Жорстка спорідненість може бути задана (маска спорідненості змінена із програми) за допомогою функцій SetProcessAffinityMask() (для процесу) або SetThreadAffinityMask() (для окремого потоку).

```
DWORD mask_proc = 2;  // другий процесор (маска 000...0010)
SetProcessAffinityMask(GetCurrentProcess(), &mask_proc)
```

Для визначення поточного значення маски спорідненості процесу використовують функцію GetProcessAffinityMask().

```
DWORD mask_proc, mask_sys;
// у масці mask_sys увімкнуто біти для всіх доступних процесорів
```

```
GetProcessAffinityMask(GetCurrentProcess(), &mask_proc, &mask_sys);
printf("маска процесу: %08lx, системна маска: %08lx\n",
    mask_proc, mask_sys);
```

Ідеальний процесор вибирають випадково під час створення потоку. Щоб змінити його із застосування, потрібно використати функцію `SetThreadIdealProcessor()`.

```
SetThreadIdealProcessor(GetCurrentThread(), 1);  // другий процесор
```

Підтримка NUMA-систем

Windows XP, як і Linux, підтримує внутрішнє відображення топології вузлів і використовує його під час планування потоків. Win32 API містить ряд функцій, що дають змогу отримати інформацію про поточну топологію (наприклад, функція `GetNumaProcessorNode()` повертає номер вузла для заданого процесора). Цю інформацію можна застосувати для оптимізації використання локальної пам'яті, наприклад шляхом задання маски спорідненості, що включає всі процесори одного вузла.

20.2. Принципи розробки розподілених систем

У цьому розділі йтиметься про базові технології розробки розподілених систем. До них належать передавання повідомлень і віддалені виклики процедур (RPC). Передавання повідомлень, а також базові принципи RPC розглянуто в розділі 6. У розділах 20.2.1–20.2.3 висвітлюватимуться особливості застосування RPC для розробки розподілених застосувань.

20.2.1. Віддалені виклики процедур

RPC відрізняється від сокетів тим, що в разі використання цієї технології мережа є прозорою для програміста. Ця прозорість виявляється в тому, що код клієнтських застосувань і серверних процедур, які взаємодіють за допомогою RPC, певною мірою виглядає так, ніби мережі немає; фактично його не можна відрізнити від коду, котрий використовує локальні процедури. Покажемо, внаслідок чого вдається досягти такої прозорості.

Заглушки

Основна ідея, на якій ґрунтуються RPC, полягає в тому, що клієнт і сервер пов'язані не прямо, а через *заглушки* (stubs) – спеціальні програмні модулі, відповідальні за мережну взаємодію. Розглянемо, які заглушки беруть участь у виклику віддаленої процедури.

1. Під час звертання до віддаленої процедури клієнтський процес використовує локальні домовленості про виклик (аналогічні до виклику локальної процедури), але насправді звертається до *клієнтської заглушки* (client stub), що маршалізує параметри і пересилає їх серверу. Під *маршалізацією* (marshaling) розуміють упакування даних у формат, придатний для передавання мережею (із мережним порядком байтів тощо), під *демаршалізацією* (demarshaling) – їхнє зворотне перетворення.
2. На сервері дані приймає *серверна заглушка* (server stub), демаршалізує параметри і викликає віддалену процедуру. Вона також буде викликана із локальними

домовленостями про виклик, її код не має інформації про те, що виклик був віддаленим. Після цього серверна заглушка маршалізує повернені значення і передає їх назад клієнтові.

3. На клієнті повернені значення демаршалізує клієнтська заглушка і передає клієнтському застосуванню аналогічно до даних локальної процедури.

Основною особливістю заглушок є те, що вони мають ті самі інтерфейси, що й відповідні процедури. Клієнтська заглушка виглядає для застосування-клієнта як локальна процедура з одними й тими самими параметрами; серверна заглушка виглядає для віддаленої процедури як локальний клієнт.

Мова IDL

Для автоматизації створення заглушок у RPC використовують опис інтерфейсів процедур спеціальною *мовою опису інтерфейсів* (Interface Definition Language, IDL). Кожну процедуру, яку потрібно викликати віддалено, необхідно описати на IDL, задавши ім'я процедури, кількість і типи параметрів, а також тип повернення. Файл з IDL-кодом, який містить оголошення таких процедур, обробляють IDL-компілятором, що генерує заглушки для клієнта і сервера.

Архітектура віддаленого виклику процедур показана на рис. 20.2.

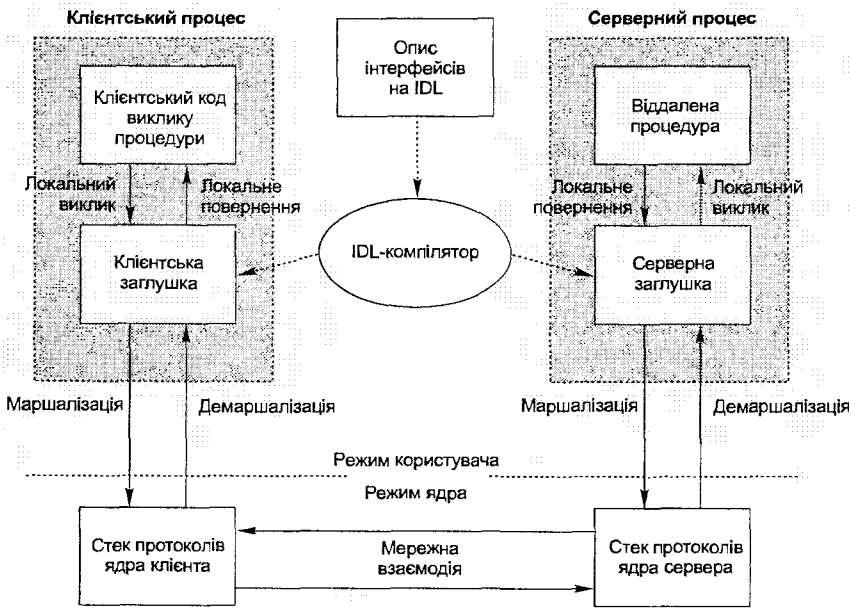

Рис. 20.2. Віддалений виклик процедур

Є різні реалізації IDL, що втілюють загальні принципи цієї мови.

1. Процедури на IDL об'єднують в інтерфейси. У цій мові під інтерфейсом розуміють набір заголовків процедур і типів, пов'язаних спільним призначенням. Інтерфейс має ім'я і номер версії. Кожний інтерфейс внаслідок обробки IDL-компілятором перетворюється на пару заглушок — клієнтську і серверну.

2. IDL є декларативною мовою, вона призначена виключно для опису інтерфейсів (фактично це зводиться до опису заголовків процедур і типів даних). Виконуваних операторів у ній немає.
3. Для забезпечення зв'язку між неоднорідними системами в IDL має бути зафіксовано набір базових типів (наприклад, може бути прийнято, що int завжди буде завдовжки 4 байти тощо).
4. Параметри процедур в IDL мають режими. Режим параметра визначає напрямок пересилання відповідних даних. Для вхідних параметрів дані передають від клієнта серверу, для вихідних — від сервера клієнтові, для параметрів змішаного режиму — в обидва боки.

20.2.2. Використання Sun RPC

Однією із найрозповсюдженіших реалізацій RPC-технології для UNIX-систем є Sun RPC [37, 52]. До її особливостей належить реалізація на основі TCP/IP, незалежність від відображення даних на клієнті та сервері (за перетворення між типами відповідає бібліотека часу виконання RPC).

Розробка файла специфікацій RPC

У документації Sun RPC не вживають терміна IDL, однак аналоги IDL-файлів і IDL-компілятора є. IDL-файлами виступають *файли специфікацій RPC* (із розширенням .x), які обробляє спеціальна утиліта rpcgen.

Насамперед у файлі специфікацій RPC задають типи даних, які використовують як параметри. Частина типів відповідає стандартним типам мови C, наприклад, int або long, рядковий тип визначають як string, максимальну довжину рядка задають у кутових дужках. Складні типи даних задають структурами (теж визначеними за правилами мови C). Часто простим типам ставлять у відповідність структури, що містять єдине поле.

Крім того, у цьому файлі задають специфікацію RPC-застосування. Вона багато в чому подібна до опису інтерфейсу (набір оголошень процедур, об'єднаних спільним іменем). Кожній процедурі присвоюють номер, унікальний у межах специфікації, всьому застосуванню відповідає номер версії та номер застосування (program number), що є 32-бітним числом, яке унікально ідентифікує RPC-сервер у рамках системи. Для серверів користувача цей номер має бути в діапазоні 0x20000000–0x3FFFFFFF.

Наведемо приклад файла специфікацій RPC для задання однієї процедури з одним параметром (припустимо, що цей файл має назву myrpc.x).

```
/* тип даних для параметра процедури */
struct msg {
    string text ;
};
/* специфікація застосування */
program HELLO_PROG {      /* ім'я застосування */
    version HELLO_VER {   /* ім'я версії */
        /* оголошення процедури із заданням номера */
        void SAYHELLO(msg) = 1;
    } = 1;                /* номер версії */
} = 20000010;             /* номер застосування */
```

Ім'я застосування і версії, а також імена процедур прийнято задавати у верхньому регістрі. Під час генерації заглушок ім'я процедури переводиться до нижнього регістра.

Файл специфікацій RPC має оброблятися утилітою rpcgen:

```
$ rpcgen myrpc.x
```

Унаслідок цього на основі myrpc.x утворяться такі файли:

- myrpc.h — оголошення віддалених процедур і типів, які будуть використані у застосуваннях;
- myrpc_clnt.c, myrpc_svc.c — коди клієнтської та серверної заглушок;
- myrpc_xdr.c — код перетворення типів у формат зовнішнього відображення (XDR), придатний для пересилання мережею (цей файл компонується разом із клієнтом і сервером).

Служба відображення портів Sun RPC

Ця служба є деяким аналогом служби іменування. Вона відповідає за відображення між номерами застосувань і TCP або UDP-портами та реалізована як фоновий процес, який звичайно називають portmapper.

Реєструють застосування із використанням його номера (це зазвичай відбувається під час запуску застосування). Під час реєстрації також задають номер порту; у більшості випадків автоматично вибирають деякий вільний порт, який далі використовуватиме це серверне застосування.

Перед викликом віддаленої процедури клієнт має зв'язатися зі службою відображення портів на відповідному хості (із використанням наперед відомого порту з номером 111), передати їй номер застосування і отримати у відповідь номер порту, через який можна обмінюватися даними із цим застосуванням.

Отримати інформацію про зареєстровані застосування можна за допомогою утиліти rpcinfo.

```
$ rpcinfo -p ім'я_хоста
```

Розробка віддалених процедур

Прототип реалізації віддаленої процедури відрізняється від її прототипу, оголошеного у файлі специфікації:

- до імені процедури додають суфікс _номер-версії_svc, наприклад, процедура sayhello() має бути реалізована як sayhello_1_svc();
- усі параметри і повернене значення замінюють відповідними покажчиками (string при цьому відповідає char *);
- як додатковий останній параметр задають покажчик на структуру svc_req, яку передає у процедуру бібліотека часу виконання і яка містить інформацію про контекст виклику.

Наведемо приклад коду віддаленої процедури:

```
#include "myrpc.h"
void * sayhello_1_svc(msg *argp, struct svc_req *rqstp) {
    static char *result = { 0 };
    printf("від клієнта: %s\n", argp->text);
    return (void *) result;
}
```

Розробка сервера

Особливістю Sun RPC є те, що у разі використання утиліти rpcgen жодних додаткових кроків для розробки сервера, крім створення коду віддалених процедур, робити не потрібно. Код функції main() для сервера, поряд із кодом серверної заглушки, генерує rpcgen і поміщає у файл myhello_svc.c. Цей код автоматично реєструє сервер у службі відображення портів, переводить його у фоновий режим і переходить до очікування з'єднань.

Під час компіляції та компонування сервера потрібно використовувати файли із визначеннями збережених процедур, файл серверної заглушки myhello_svc.c і файл XDR-перетворень myhello_xdr.c.

Розробка клієнта

Перед викликом процедури на віддаленому сервері в коді клієнта необхідно отримати клієнтський дескриптор, що буде використаний далі для виклику збережених процедур із сервера. Цей дескриптор є покажчиком на спеціальну структуру CLIENT. Для його отримання використовують функцію clnt_create()

```
#include <rpc/rpc.h>
CLIENT *clnt_create(const char *host, unsigned long pnum,
    unsigned long ver, const char *protocol);
```

де: host — ім'я або IP-адреса віддаленого хоста;

pnum — номер застосування (у заголовному файлі, згенерованому rpcgen, для нього визначають відповідну константу, наприклад HELLO_PROG);

ver — номер версії (для нього визначають аналогічну константу, наприклад HELLO_VER);

protocol — рядок із назвою протоколу ("tcp", "udp").

Заголовний файл <rpc/rpc.h> буде автоматично підключений у myrpc.h, тому явно підключати його не обов'язково.

```
#include "myrpc.h"
CLIENT *cl;
cl = clnt_create("myserver", HELLO_PROG, HELLO_VER, "tcp");
```

Після отримання клієнтського дескриптора можна викликати віддалену процедуру. Для цього викликають процедуру ім'я_номер_версії(): для процедури sayhello() треба викликати sayhello_1(). Усі аргументи передають у цю функцію як покажчики, останнім аргументом додатково задають клієнтський дескриптор.

```
struct msg hello = { "hello world" };
void *res = sayhello_1(&hello, cl);
```

Виклик віддаленої процедури може повернути NULL: це означає, що сталася помилка. Рядок з інформацією про помилку в цьому разі повертає функція clnt_sperror(), що першим параметром приймає клієнтський дескриптор, а другим — ім'я віддаленого хоста:

```
if (res == NULL)
    printf("Помилка: %s\n", clnt_sperror(cl, "myserver"));
```

Забезпечення безпеки даних у разі використання Sun RPC

Для забезпечення безпеки даних у разі використання RPC застосовують аутентифікацію RPC-клієнтів перед доступом до сервера. Є кілька рівнів такої аутентифікації.

Рівень AUTH_NONE (використовуваний за замовчуванням) означає, що аутентифікації не виконують зовсім. Відповідно до нього будь-який клієнт у мережі, що може відсилати пакети RPC-серверу, має змогу викликати будь-яку реалізовану ним процедуру. Цей рівень не забезпечує ніякого захисту і не рекомендований до використання.

Рівень AUTH_UNIX означає, що кожний RPC-запит супроводжується ідентифікатором користувача (uid) і набором ідентифікаторів груп (gid). Мається на увазі, що ці ідентифікатори відповідають користувачу, який запустив клієнтське застосування, і що сервер довіряє цьому користувачу.

Для використання такої аутентифікації на клієнті після виклику clnt_create() необхідно виконати код

```
// CLIENT *cl = clnt_create(...)
auth_destroy(cl->cl_auth);
cl->cl_auth = authsys_create_default();  // задання AUTH_UNIX
```

Цей рівень теж не зовсім відповідає сучасним уявленням про мережну безпеку, оскільки зловмисник може створювати і відсилати RPC-пакети із довільними значеннями uid і gid, і їхнє авторство не може бути перевірене сервером. Наприклад, коли відомо, який користувач потрібен для виконання необхідних процедур, і в мережі є комп'ютер, на якому зловмисник має права root, він може створити користувача із необхідним uid і виконувати RPC-запити клієнтським процесом, запущеним під цим користувачем. Такі запити будуть успішно виконані, хоча пароль потрібного користувача RPC-сервера зловмисникові невідомий.

Рівень AUTH_DES використовує гібридну криптосистему для організації захищеного каналу зв'язку для RPC-виклику. Реалізація такого рівня аутентифікації відома як Secure RPC.

20.2.3. Використання Microsoft RPC

Розробка застосувань із використанням Microsoft RPC ґрунтується на тих самих принципах, що і Sun RPC, але має деякі особливості [50]. Насамперед ця технологія може використовувати різні базові засоби зв'язку (зокрема TCP/IP та поіменовані канали). Крім того, можлива розробка клієнтського застосування без явного задання з'єднання із сервером.

Розробка IDL-файла

Розробку застосування починають з опису його інтерфейсів на IDL. Кожному інтерфейсу ставлять у відповідність *універсальний унікальний ідентифікатор* (UUID) – 128-бітне число, генероване за допомогою спеціального алгоритму, що забезпечує високий ступінь унікальності. За цим ідентифікатором RPC-клієнти ідентифікують інтерфейси, експортовані серверами. Для створення UUID використовують утиліту uuidgen.

Кожний інтерфейс супроводжують атрибути, перелічені у квадратних дужках перед ключовим словом `interface`:

- `uuid` – UUID для цього інтерфейсу;
- `version` – версія інтерфейсу;
- `auto_handle` – означає, що клієнтське застосування не повинне явно задавати код для встановлення з'єднання із сервером – це буде зроблено із коду заглушки (є й інші способи організації зв'язку, наприклад, за наявності атрибута `implicit_handle` зв'язок має бути встановлений явно).

Кожен параметр інтерфейсу теж супроводжують атрибути:

- `in, out` – режим параметра (`in` – вхідний, `out` – вихідний);
- `string` – параметр треба розглядати як рядок символів.

Наведемо приклад IDL-файла, що задає один інтерфейс з однією процедурою. Далі вважатимемо, що він називається `myrpc.idl`.

```
[uuid(c0579cff-e76a-417d-878b-1195d366385), version(1.0), auto_handle]
interface ihello {   /* визначення інтерфейсу ihello */
    void say_msg([in,string] unsigned char* msg);
}
```

IDL-файл обробляють IDL-компілятором (`midl`):

```
midl.exe /app_config myrpc.idl
```

Параметр `/app_config` задають у разі, коли в IDL-файлі присутні атрибути, що стосуються всього застосування (у нашому випадку це `auto_handle`). Якщо його не вказати, такі атрибути задаються в окремому файлі з розширенням **.acf**.

Внаслідок обробки IDL-файла створяться файли заглушок для клієнта і сервера (**myrpc_c.c** і **myrpc_s.c**), які треба скомпонувати у відповідні виконувані файли, а також заголовний файл **myrpc.h**, що має бути підключений у вихідні файли клієнта і сервера.

Розробка віддалених процедур

Віддалені процедури мають такий вигляд, як і звичайні. Зазначимо, що типи даних IDL відповідають типам мови C (а не типам Win32 API, таким як `DWORD`).

```
void say_msg(unsigned char* msg) {    // код віддаленої процедури
    printf("від клієнта: %s\n", msg);
}
```

Розробка сервера

У коді сервера насамперед потрібно вказати бібліотеці підтримки RPC, який протокол збирається використати сервер і як ідентифікувати сервер відповідно до цього протоколу (яка його кінцева точка – endpoint).

```
RPC_STATUS RPC_ENTRY RpcServerUseProtseqEp(unsigned char *prot,
unsigned int max_calls, unsigned char *endpoint, void *sec_desc);
```

де: `prot` – рядок, що визначає протокол (`"ncacn_ip_tcp"` – TCP/IP, `"ncacn_np"` – поіменовані канали);

max_calls — максимальна кількість з'єднань із сервером (значення за замовчуванням задають як RPC_C_LISTEN_MAX_CALLS_DEFAULT);

endpoint — рядок, що визначає кінцеву точку (для TCP/IP він задає порт, для поіменованих каналів — ім'я каналу).

Ця функція повертає статус RPC-виклику; якщо він дорівнює 0 (RPC_S_OK) — виклик завершився успішно. Аналогічний код повертають і інші RPC-функції.

```
// сервер використовує TCP/IP, прослуховує порт 5000
RpcServerUseProtseqEp("ncacn_ip_tcp", 5, "5000", NULL);
```

Після задання протоколу необхідно зареєструвати інтерфейси у бібліотеці підтримки RPC для того, щоб клієнти могли його знаходити:

```
// для кожного інтерфейсу з IDL-файла
RpcServerRegisterIf(ihello_v1_0_s_ifspec, NULL, NULL);
```

Першим параметром задають структуру визначення інтерфейсу, яку генерує midl. Ім'я такої структури будують на підставі імені інтерфейсу і його версії: ім'я_v1_0_s_ifspec.

Тепер клієнти зможуть знайти сервер, і можна перейти в режим очікування з'єднань:

```
// очікування з'єднань від клієнтів
RpcServerListen(1, 5, FALSE);
```

Перед завершенням роботи сервер має скасувати реєстрацію своїх інтерфейсів у бібліотеці підтримки

```
RpcServerUnregisterIf(NULL, NULL, FALSE);
```

Розробка клієнта

Як зазначалося, у разі використання в IDL-файлі атрибута auto_handle у коді клієнта не потрібно явно задавати з'єднання із сервером. У цьому разі код RPC-клієнта може взагалі не відрізнятися від коду застосування, що викликає локальні процедури. Але оскільки RPC-клієнт звертається до віддалених серверів, помилки, які можуть виникати, мають оброблятися особливим чином. Для організації обробки помилок є набір спеціальних макросів, які реалізують обробку виняткових ситуацій у стилі мови C++: RpcTryExcept задає блок для перевірки, RpcExcept — оброблювач виняткової ситуації, RpcEndExcept завершує код обробки. Для визначення коду помилки в тілі оброблювача використовують функцію RpcExceptionCode().

```
RpcTryExcept {
    say_msg("hello");   // виклик віддаленої процедури
}
RpcExcept(1) {
    printf("помилка RPC-виклику з кодом %d\n", RpcExceptionCode());
}
RpcEndExcept;
```

Функції динамічного керування пам'яттю

Розробники RPC-застосувань мають включити у проект код двох функцій, що забезпечують динамічний розподіл пам'яті: midl_user_allocate() і midl_user_free().

Вони автоматично викликаються із коду RPC-бібліотеки. Для їхньої реалізації зазвичай достатньо скористатися С-функціями `malloc()` і `free()`.

```
void __RPC_FAR * __RPC_USER midl_user_allocate(size_t size) {
    return malloc(size);
}
void __RPC_USER midl_user_free(void* p) {
    free(p);
}
```

20.2.4. Обробка помилок і координація в розподілених системах

Як бачимо, технологія RPC — це прозорий для користувача спосіб перетворення локальних програм у розподілені. Деякі реалізації RPC дають змогу для розв'язання цієї задачі обмежитися мінімальними змінами програмного коду. Виникає запитання: чи завжди коректний такий «автоматичний» підхід? Під час вивчення цього розділу побачимо, що це далеко не так. Можна сказати, що рівень абстракції, наданий RPC, не завжди відповідає специфіці розподілених систем.

Це пов'язано з тим, що структура застосувань, розроблених із використанням RPC, розрахована на роботу за умов успішного функціонування мережі, при цьому обробку помилок виконують майже так само, як і для локальних застосувань. Віддалена процедура, подібно до локальної, повертає простий код помилки («мережна помилка», «немає зв'язку» тощо), при цьому немає можливості дослідити, які компоненти розподіленої системи виявилися джерелом проблеми.

Насправді помилки в розподілених системах значно відрізняються за своїм характером від локальних помилок, оскільки їхнім джерелом є взаємодія цілого набору ненадійних компонентів (мережних з'єднань, проміжних вузлів тощо). Крім того, можлива поява часткових помилок, за яких деяку частину дій виконують коректно, а якусь — ні.

Сформулюємо головне правило обробки помилок у розподіленій системі, що не виконується у разі використання RPC: поведінка розподіленої системи при виникненні помилок має бути досліджена так само ретельно, як і коректна поведінка системи. У цьому пункті розглянемо деякі особливості такої поведінки і способи її дослідження. Докладніше про це питання у [45].

Задача розподіленої координації

Цю задачу формулюють так: чи можливо надійно скоординувати дії на двох різних комп'ютерах (наприклад, виконання деякого коду в один і той самий час, виконання дії тільки один раз, атомарну зміну стану на двох різних машинах тощо)?

До дій, що вимагають такої синхронізації, належать, наприклад:

- атомарне переміщення файла із сервера S_1 на сервер S_2;
- атомарне переміщення суми грошей із рахунку в одному банку на рахунок в іншому.

Головна проблема, що виникає при цьому, пов'язана з тим, що мережа, яка зв'язує ці комп'ютери, є ненадійною:

- повідомлення можуть бути втрачені;
- комп'ютери можуть бути вимкнені або вийти з ладу.

З урахуванням ненадійності мережі можна уточнити формулювання цієї задачі.

Чи можна використати обмін повідомленнями через ненадійну мережу для надійної синхронізації двох комп'ютерів (щоб вони гарантовано могли виконувати ту саму операцію одночасно)?

Відповідь на це запитання є однозначно негативною навіть тоді, коли всі повідомлення будуть доставлені й жоден комп'ютер не вийде з ладу.

Цю проблему звичайно ілюструють парадоксом візантійських генералів. Припустимо, що два генерали на чолі армій перебувають на деякій відстані один від одного, тому вони можуть обмінюватися повідомленнями тільки через посильних, причому під час доставлення повідомлення посильних можуть полонити. Задачею є погодити час одночасної атаки на ворога (спільних сил достатньо для перемоги, сил кожної армії окремо – ні). Розглянемо протокол обміну повідомленнями, який можна використати в даному випадку.

1. Генерал G_A відсилає повідомлення M_{A1} із точним часом атаки.
2. Генерал G_B отримує M_{A1}, відсилає своє повідомлення M_{B1} із висловленням згоди і починає очікувати підтвердження отримання M_{B1}.
3. Генерал G_A отримує M_{B1}, відсилає підтвердження його отримання (M_{A2}) і починає очікувати підтвердження отримання M_{A2}.

Подібний обмін повідомленнями може тривати нескінченно. Очевидно, що погодити час атаки генералам не вдасться (останнє повідомлення увесь час залишатиметься непідтвердженим). Доведено, що розв'язків ця задача не має.

Далі в цьому пункті розглянемо протоколи розподіленої координації, що не потребують одночасного виконання операцій різними сторонами.

Протокол запиту-квітування

Проаналізуємо особливості обміну повідомленнями в мережі за наявності помилок. Припустимо, що трапляються тільки помилки передавання даних (втрати повідомлень), учасники протоколу не виходять із ладу.

Для обміну повідомленнями за таких умов із гарантією одноразового доставлення призначений протокол запиту-квітування (request/acknowledge protocol).

1. Відправник пересилає одержувачеві повідомлення і встановлює таймер.
2. Одержувач отримує повідомлення і відсилає підтвердження отримання.
3. Відправник отримує підтвердження і скидає значення таймера.
4. У разі вичерпання часу, заданого таймером, кроки протоколу повторюють.

Цей протокол гарантує, що повідомлення буде доставлене хоча б один раз за умови, що учасники не виходять із ладу. При цьому одержувач може отримати повідомлення кілька разів (у разі втрати підтвердження). Щоб мати можливість відкидати дублікати, повідомлення можна супроводжувати унікальними номерами.

Протокол запиту-квітування мало пристосований до роботи за умов виходу з ладу його учасників. Наприклад, у разі краху відправника після відновлення він не може отримати інформацію про те, отримав одержувач повідомлення чи ні (підтвердження до нього дійти не зможе). У разі краху одержувача значно ускладнюється робота із дублікатами.

Для координації взаємодії ненадійних учасників потрібні більш складні протоколи. Одним із них є протокол двофазового підтвердження.

Двофазове підтвердження

Оскільки одночасного виконання дій на віддалених комп'ютерах відповідно до парадоксу генералів домогтися неможливо, для розподіленої координації дій використовують методи, у яких допустиме виконання дій у різний час. Розглянемо один із них – *двофазове підтвердження* (two-phase commit, 2PC) [4].

У цьому разі дві сторони виконують дії, оформлені у вигляді транзакцій – атомарних операцій, які виконуються повністю (підтверджуються, commit) або не виконуються зовсім (відкочуються, rollback).

Прикладом розподіленої транзакції може бути перенесення файла із сервера S_1 на сервер S_2. Якщо виконання цієї дії переривається, за відсутності атомарності може виявитися, що файл був вилучений із сервера S_1, але при цьому не був перенесений на сервер S_2.

Для виконання протоколу двофазового підтвердження необхідно на кожному комп'ютері підтримувати журнал транзакцій (transaction log), у якому відстежувати, відбулося підтвердження чи ні. Крім того, один із вузлів мережі має відігравати роль координатора транзакцій. Основним завданням такого координатора є збереження атомарності кожної транзакції у розподіленій системі.

Першим етапом протоколу є етап запитів координатора.

1. Координатор відсилає всім учасникам запит. Він містить опис дії, яка має виконуватися, наприклад «вилучити файл a.txt із кореневого каталогу». Перед відсиланням запиту інформацію про нього зберігають у журналі транзакцій координатора.
2. Учасники отримують запит і виконують транзакцію локально (наприклад, намагаються вилучити файл). Інформацію про результат цієї дії відображають у вигляді повідомлення (M_{OK} – успіх, M_{ABORT} – невдача), яке зберігають у журналі транзакцій кожного учасника і відсилають координаторові. Цей етап завершений, коли координатор отримає повідомлення від всіх учасників (або мине максимальний час очікування). Зазначимо, що для жодної транзакції в конкретний момент ще не було виконано ні підтвердження, ні відкату (наприклад, інформацію про вилучення файла зберігають лише у пам'яті).

   Другим етапом є етап ухвалення рішення координатором.
3. Спочатку координатор ухвалює рішення щодо підтвердження або відкату розподіленої транзакції. На цьому кроці можуть виникнути дві ситуації:
   - координатор отримує повідомлення M_{ABORT} хоча б від одного учасника. Після цього він створює повідомлення $M_{GLOBAL\_ROLLBACK}$, зберігає його в журналі та відсилає всім учасникам;
   - координатор отримує повідомлення M_{OK} від кожного учасника. Після цього він створює повідомлення M_{GLOBAL_COMMIT}, зберігає його в журналі та відсилає всім учасникам.
4. Після отримання повідомлення від координатора кожен учасник зберігає його у своєму журналі та виконує відкат або підтвердження транзакції залежно від типу повідомлення. Для цього прикладу відкат може зводитися до повного скасування операції вилучення файла, підтвердження – до вилучення файла з диска.

Розглянемо виконання цього протоколу за умов мережних помилок. Спочатку зупинимося на випадках виходу з ладу учасників протоколу.

- Якщо учасник вийде з ладу на кроці 2 (під час виконання транзакції), координатор не отримає повідомлення про завершення транзакції упродовж максимального часу очікування і відкотить транзакцію, відіславши учасникам повідомлення $M_{GLOBAL\_ROLLBACK}$.
- Якщо координатор вийде з ладу на кроці 3, після відновлення він має звернутися до свого журналу транзакцій. Коли в ньому немає жодного повідомлення M_{GLOBAL_XXX} або є повідомлення $M_{GLOBAL\_ROLLBACK}$, то всім учасникам відсилають повідомлення про відкат. Коли в журналі є M_{GLOBAL_COMMIT}, учасникам відсилають повідомлення про підтвердження.
- У разі виходу учасника з ладу на кроці 4 після відновлення він має звернутися до координатора за інформацією і виконати відповідно підтвердження або відкат.

До недоліків протоколу 2PC належать високі вимоги до надійності координатора. Якщо він вийде з ладу на кроці 3 і не відновить свою роботу, всі учасники можуть бути заблоковані.

Це, однак, відбувається не завжди. У разі отримання деякими учасниками повідомлення від координатора на кроці 4, а деякими — ні, часом є можливість завершити транзакцію, звернувшись за інформацією до інших учасників.

- Коли хоча б один із учасників отримав M_{GLOBAL_XXX}, необхідно виконати дію, що відповідає цьому повідомленню.
- Коли хоча б один із учасників відіслав M_{ABORT}, транзакцію необхідно відкотити.

Якщо ж всі учасники відіслали M_{OK}, але ніхто з них не отримав M_{GLOBAL_XXX}, рішення не може бути прийнято, оскільки координатор міг зберегти в журналі (але не встиг відіслати) $M_{GLOBAL\_ROLLBACK}$ через свою внутрішню помилку. Така ситуація і спричиняє блокування всіх учасників. Під час реалізації 2PC потрібно завжди враховувати можливість такого блокування.

20.3. Розподілені файлові системи

Розподілена файлова система (РФС) дає можливість застосуванням звертатися до файлів, що перебувають на диску віддаленого комп'ютера. Загалом до таких систем можна віднести будь-які засоби доступу до віддалених файлів; наприклад, HTTP фактично можна розглядати як базовий протокол для реалізації РФС.

У цьому розділі обмежимося розглядом РФС, які володіють властивістю прозорості доступу. Під час роботи із такою системою з усіма файлами (локальні вони чи віддалені) працюють однаково, особливості реалізації віддаленого доступу обробляють на рівні файлової системи. Фактично розподілена файлова система із прозорістю доступу приховує від користувача наявність мережі.

20.3.1. Організація розподілених файлових систем

Цей розділ присвячено опису загальних особливостей організації РФС.

Керування іменами в РФС

Є два основні підходи до реалізації керування іменами в розподілених файлових системах із прозорістю доступу. Основні відмінності між ними полягають у вигляді файлової системи віддаленого комп'ютера на комп'ютері-клієнті.

Перший підхід ґрунтується на тому, що відображення файлової системи для різних клієнтів може виглядати по-різному. Так відбувається у разі віддаленого монтування файлових систем. У цьому разі перед використанням віддаленої файлової системи її потрібно змонтувати у підкаталог файлової системи клієнта. Різні клієнти можуть монтувати ту саму систему в різні каталоги. Такий підхід реалізувати досить просто (немає потреби відстежувати, яким чином різні клієнти отримують доступ до файлів), але керування системою стає складнішим (у файловій системі клієнта виявляються «перемішаними» локальні та віддалені каталоги, до одного й того самого файла звертаються із використанням різних шляхів тощо).

У разі реалізації другого підходу файлова система визначає єдиний простір імен, що матиме однаковий вигляд для всіх її клієнтів. Таке єдине відображення дає змогу спростити використання системи. Цей підхід складніший у реалізації, але є найбільш зручним для користувачів.

Важливою додатковою властивістю, що може бути реалізована у РФС, є незалежність від місця розташування (location independence). Якщо її підтримують, шлях до файла залишається незмінним у разі його фізичного переміщення (наприклад, з одного сервера на інший). Ця властивість доволі складна в реалізації і рідко підтримується в сучасних РФС.

Реалізація віддаленого доступу до файлів

Припустимо, що клієнт звернувся до віддаленого файла, а засоби керування іменами дали змогу знайти його фізичне місце розташування. Після цього необхідно звернутися мережею до віддаленого файла, обробити його і повернути результат клієнтові. Розглянемо підходи до реалізації такого віддаленого доступу до файла.

Найпростішим підходом є передавання кожного запиту на сервер у вигляді окремого віддаленого виклику процедури. При цьому жодного кешування даних не роблять, кожний запит вимагає передавання даних мережею. Це віддалене обслуговування (remote service). Основними його перевагами є простота реалізації та відсутність суперечностей (всі дані негайно зберігаються на диску сервера), а головним недоліком — низька продуктивність, оскільки кожна файлова операція вимагає пересилання даних мережею та звертання до диска на сервері.

Для підвищення продуктивності використовують кешування даних. У даному випадку локальні кеші розташовані на клієнтах (звичайно в їхній основній пам'яті), а основною копією даних є копія даних на сервері. При цьому головною проблемою, як і для багатопроцесорних систем, є необхідність підтримки когерентності локальних кешів.

Наведемо два способи підтримки погодженості даних у локальних кешах.

Перший зводиться до того, що в разі будь-якої зміни даних в одному із кешів їх негайно записують на диск відповідно до методу віддаленого обслуговування, при цьому очищують усі інші кеші. Фактично так реалізований кеш із наскрізним записом (див. пункт 12.3.2). Операції читання даних продовжують обслуговувати із кеша.

Другим способом є використання сеансової семантики (session semantics). При цьому всі зміни, які вносять у файл після його відкриття клієнтським процесом (упродовж сеансу роботи клієнта із цим файлом), будуть доступні тільки цьому процесові. Після закриття файла його змінену копію пересилають на сервер, і вона стає доступною іншим клієнтським процесам.

20.3.2. Файлова система NFS

Найвідомішим прикладом реалізації розподілених файлових систем є Network File System (*NFS*) [44]. Сьогодні NFS є стандартною файловою системою більшості UNIX-сумісних ОС.

Принцип дії NFS

Файлова система NFS об'єднує окремі каталоги файлових систем віддалених комп'ютерів у єдине дерево каталогів локального комп'ютера. Таке об'єднання є прозорим для користувачів – з їхнього погляду остаточне дерево каталогів не відрізняється від локальної файлової системи UNIX.

Архітектура цієї файлової системи реалізована на базі взаємодії між NFS-клієнтами і NFS-серверами. Зазначимо, що комп'ютер може одночасно відігравати роль клієнта і сервера.

1. Для того щоб NFS-клієнт міг отримати доступ до каталогу локальної файлової системи NFS-сервера, цей каталог потрібно експортувати. Для цього необхідно додати відповідний шлях у список експорту, який зберігають на сервері, зазвичай у спеціальному файлі /etc/exports. Цей список зчитує ядро ОС під час завантаження системи.
2. Для доступу до експортованих каталогів сервера NFS-клієнти мають їх вмонтувати в каталоги своєї локальної файлової системи аналогічно до того, як монтують файлові системи. Різні клієнти можуть монтувати віддалений каталог у різні точки своїх файлових систем, причому сервер не має інформації про точки монтування його каталогів.

Віддалений каталог внаслідок монтування може навіть стати кореневим каталогом файлової системи клієнта. Так реалізовано роботу бездискових робочих станцій, які завантажуються через мережу.

NFS використовує протокол монтування і протокол NFS. Як базові компоненти обидва протоколи використовують віддалені виклики процедур відповідно до інтерфейсу Sun RPC.

Протокол монтування

Протокол монтування використовують для задання вихідного логічного з'єднання між сервером і клієнтом під час операції монтування. Розглянемо кроки цього протоколу.

1. Клієнт виконує RPC-запит MNT для виклику процедури монтування. Параметром цієї процедури на сервер передають шлях до віддаленого каталогу.
2. Якщо цей каталог був експортований і обмеження доступу дають змогу виконати монтування, клієнтові повертають файловий дескриптор, що містить всю інформацію, необхідну для доступу до файлів усередині каталогу (ідентифікатор локальної файлової системи сервера та індексний дескриптор каталогу всередині цієї файлової системи).

Є два підходи до монтування віддалених файлових систем. Перший – це явне монтування, аналогічне до монтування локальних файлових систем. Зокрема, специфікації явного монтування можуть бути додані у файл /etc/fstab. Така конфігурація, однак, призводить до проблем, якщо віддалена система недоступна в момент завантаження.

Більш розповсюдженим підходом є автоматичне монтування (automount), коли фактичне монтування каталогу виконують після першого звертання до нього.

Протокол NFS

Цей протокол визначає набір RPC-запитів, що реалізують операції із віддаленими файлами і каталогами (пошук файла в каталозі — LOOKUP, читання набору елементів каталогу — READDIR, створення і вилучення файлів і каталогів — CREATE, REMOVE, MKDIR, RMDIR, читання і записування файлів — READ і WRITE, робота з атрибутами файлів — GETATTR, SETATTR).

Серед цих NFS-запитів, однак, немає аналогів системних викликів open() і close(). Це відображає фундаментальну властивість NFS-серверів — вони не зберігають стану між звертаннями клієнтів (див. розділ 11.5.1). Такий підхід використовують через небезпеку втрати стану в разі виходу сервера з ладу (у цьому випадку подальше використання відкритих файлів стане неможливим).

Те, що NFS-протокол не зберігає стану, визначає дві важливі характеристики запитів, що входять у нього.

1. NFS-запити є незалежними. Усю інформацію, необхідну для виконання такого запиту, потрібно передавати в нього як параметри. Наприклад, запит WRITE має приймати як параметри унікальний ідентифікатор файла та зсув усередині файла.
2. Більшість NFS-запитів мають бути ідемпотентними. Це означає, що NFS-клієнт може відіслати серверу один і той самий запит кілька разів, при цьому загальний результат їхнього виконання має залишитися тим самим. Наприклад, читання дискового блока із файла (запит READ) є ідемпотентним: внаслідок виконання операції повертаються ті самі дані незалежно від того, скільки разів вона була повторена.

Не всі запити мають властивість ідемпотентності. Наприклад, повторне виконання запиту REMOVE призводитиме до спроби вилучення відсутнього файла. Для вирішення цієї проблеми сучасні реалізації NFS підтримують спеціальний кеш повторних запитів, у якому зберігають інформацію про запити, виконані останнім часом. Якщо новий запит збігається із якимось запитом із цього кеша, його не виконують.

Протокол NFS — це реалізація протоколу запиту-квітування. Якщо клієнт не отримає підтвердження виконання запиту до вичерпання часу очікування, він пересилає той самий запит повторно. Це повторюють доти, поки запит не буде виконано (при цьому час очікування підтвердження із кожною новою спробою подвоюють), після чого клієнт продовжує свою роботу, ніби й нічого не сталося (внаслідок ідемпотентності всіх відісланих запитів).

Якщо сервер вийде з ладу, клієнт повторюватиме запити в циклі і продовжувати роботу не зможе. Спосіб завершення циклу задають під час монтування каталогу. У разі жорсткого монтування (hard mount) цикл триватиме нескінченно (він може бути перерваний тільки після відновлення роботи сервера), у разі м'якого монтування (soft mount) цикл переривають після закінчення деякого часу (звичайно кількох хвилин), а клієнтові повертають помилку. За замовчуванням використовують жорстке монтування.

Кешування у NFS

Проблеми, які виникають під час реалізації кешування в NFS, наведено нижче.

Першою з них є можливість втрати даних із кеша. Зокрема у разі виходу з ладу сервера всі дані, які перебувають у його дисковому кеші, але ще не записані на диск, будуть втрачені. Вихід із ладу клієнта, з іншого боку, може спричинити втрати модифікованих даних у локальному кеші клієнта.

Для вирішення цієї проблеми у NFS застосовують кеш із наскрізним записом. Внаслідок виконання виклику `WRITE` всі модифіковані дані мають зберігатися на диску сервера до того як керування буде повернене клієнтові. Клієнт може кешувати дані локально, але після того як їх передали серверу, вважають, що вони записані на диск. Це значно зменшує продуктивність (тепер кожна операція записування спричиняє звертання до диска на сервері), але гарантує, що вихід із ладу сервера не впливатиме на відображення даних для клієнта (після повернення сервера до роботи клієнт може негайно продовжувати працювати з ним без повторного монтування файлової системи).

Другою проблемою є забезпечення когерентності кеша клієнтів. Опишемо базовий спосіб вирішення цієї проблеми у NFS.

Якщо клієнт змінює файл, дані зберігають на сервері. При цьому інші клієнти продовжують використовувати стару версію зі своїх кешів до вичерпання деякого часового проміжку (довжина якого може бути змінена адміністратором сервера; звичайно вона становить від 3 до 30 с). Після цього перевіряють, чи не змінився файл на сервері, і, якщо змінився, клієнт отримує нову копію і зберігає її в локальному кеші. Як наслідок під час роботи з NFS звичайною є ситуація, коли файли, створені одним клієнтом, залишаються невидимими для всіх інших упродовж досить тривалого часу.

Файлові блокування у NFS

Організація файлових блокувань на рівні NFS-протоколу неможлива, оскільки для цього потрібне збереження інформації на сервері. Сучасні версії NFS реалізують підтримку блокувань за допомогою окремого протоколу NLM (Network Lock Manager). Коли NFS-клієнт запроваджує або знімає файлове блокування, буде згенеровано RPC-виклик відповідно до цього протоколу.

Протокол NLM підтримує збереження стану. У зв'язку з цим його використання ускладнює обробку збоїв клієнта і сервера.

- У разі виходу з ладу сервера клієнти знімають свої блокування; це відбувається тоді, коли вони перестають отримувати від NLM-сервера повідомлення, що підтверджують його функціонування.
- У разі перезапуску клієнта він має надіслати серверу відповідне повідомлення, після чого сервер вивільняє всі його блокування. Зазначимо, що в разі аварійного завершення клієнт зазвичай не встигає відіслати таке повідомлення, і його блокування адміністратор системи має вивільняти вручну.

Безпека даних у NFS

Керування доступом у NFS ґрунтується на рівнях аутентифікації Sun RPC.

Вихідна модель керування доступом не є безпечною, оскільки вона використовує рівень аутентифікації `AUTH_UNIX`. Кожен NFS-запит супроводжують `uid` і набір `gid`, які порівнюють з атрибутами безпеки необхідного файла. У разі збігу

ідентифікаторів і наявності відповідних прав вважають, що доступ до файла дозволено. Проблеми, що виникають при цьому, описані у пункті 20.2.2.

Деякі сучасні реалізації NFS дають змогу використовувати аутентифікацію рівня AUTH_DES, таку технологію називають Secure NFS.

20.3.3. Файлова система Microsoft DFS

Раніше розглядались особливості реалізації спільного використання файлів у Windows XP. Незважаючи на те, що таке використання має багато переваг, воно не визначає РФС із прозорістю доступу.

- UNC-імена дають змогу уніфіковано звертатися до мережних ресурсів, але не приховують використання мережі. Ім'я віддаленого комп'ютера потрібне для кожного звертання до ресурсу.
- Відображення імен мережних ресурсів на імена томів приховує використання мережі, але не дозволяє створити єдину файлову систему на базі мережних ресурсів. Кожний ресурс має вигляд для користувача як окремий том, одночасно може бути доступно стільки ресурсів, скільки є вільних букв алфавіту.

Цих недоліків позбавлена розподілена файлова система DFS (Distributed File System), доступна в ОС лінії Windows XP.

Принципи організації DFS

Дана файлова система дає змогу створити єдине логічне дерево каталогів, що відображають мережні ресурси, багато в чому аналогічне до стандартної файлової системи UNIX, яка використовує NFS-розділи. Корінь дерева називають *кореневим каталогом DFS* (DFS root). Його створюють на NTFS-розділі жорсткого диска комп'ютера, який називають *DFS-сервером*. Починаючи від цього кореня, адміністратор може створювати *DFS-зв'язки* (DFS links). Кожен такий зв'язок відображає віддалений мережний ресурс (для його створення необхідно вказати UNC-ім'я цього ресурсу). Комп'ютер, що надає мережний ресурс, називають довідковим комп'ютером (referral computer). Доступ до DFS-зв'язку для клієнта аналогічний до доступу до звичайного каталогу через UNC-ім'я:

```
\\DFS-сервер\кореневий_каталог_DFS\DFS-зв'язок\ім'я_файла
```

Різні зв'язки можуть відповідати ресурсам на різних віддалених комп'ютерах, відображати різні файлові системи – для користувача або прикладної програми це не має значення – їх усіх видно як частини одного дерева каталогів (аналогічно до UNIX). Для доступу до мережного ресурсу через DFS-зв'язок достатньо, щоб клієнт, DFS-сервер і довідковий комп'ютер використовували спільний мережний протокол. Сьогодні підтримка DFS є вбудованою у стандартний CIFS-редиректор, тому Windows-системи можуть отримувати доступ до DFS-каталогів без додаткового налаштування. З іншого боку, довідкові комп'ютери теж не вимагають додаткових налаштувань (крім створення мережних ресурсів), оскільки інформація про структуру DFS зберігається лише на DFS-сервері.

Дублікатні набори і стійкість до збоїв

DFS дає змогу задавати *дублікатні набори* (replica sets) для окремих DFS-зв'язків. У такий набір може входити кілька мережних ресурсів на різних комп'ютерах (звичайно – однойменних каталогів), які підтримують у синхронізованому стані.

При цьому реалізоване балансування навантаження (у разі доступу до зв'язку випадково вибирають один із ресурсів набору, із якого і будуть отримані дані) і стійкість до збоїв (якщо вийде з ладу один ресурс набору, всі запити буде автоматично переадресовано іншим ресурсам).

Архітектура DFS

Підтримка DFS на сервері складається із фонового процесу dfssvc.exe (служби DFS) і драйвера файлової системи dfs.sys.

Служба DFS відповідає за керування основною структурою даних DFS – таблицею інформації про розділи (Partition Knowledge Table, PKT), що містить відомості про відповідність між DFS-зв'язками і фізичними серверами, на яких перебувають дані.

Драйвер DFS безпосередньо відповідає за перетворення конфігурації DFS (що перебуває у PKT) на інформацію про віддалений ресурс, яку повертають клієнтові.

Поняття посилання. Пошук інформації в DFS

Окремий елемент PKT називають посиланням (referral). Він зберігає інформацію про DFS-зв'язок разом із UNC-іменем віддаленого ресурсу або набором його дублікатів.

DFS-клієнт містить локальний PKT-кеш, призначений для зберігання посилань. У разі доступу до DFS-каталогу клієнт перевіряє, чи не збережене у його PKT-кеші посилання, що відповідає цьому каталогу. Якщо такого посилання немає, клієнт запитує його у DFS-сервера. Сервер відшукує посилання у PKT і повертає клієнтові, який використовує відповідне UNC-ім'я для доступу до ресурсу, а саме посилання заносить у кеш. Якщо посилання було знайдене в кеші, UNC-ім'я із нього використовують негайно без звертання до сервера. Коли посилання містить набір дублікатів, DFS-сервер визначає IP-адресу клієнта і випадково (але з деякою перевагою для ресурсів, ближчих до клієнта) вибирає мережний ресурс із цього набору.

Для керування DFS-службою використовують RPC, для обміну даними – стандартні засоби CIFS.

20.4. Сучасні архітектури розподілених систем

У цьому розділі йтиметься про дві найважливіші архітектури розподілених систем: *кластерну* та *обчислювальної мережі* (grid-архітектура). Спільним для них є те, що вони дають змогу об'єднати обчислювальні потужності багатьох комп'ютерів для розв'язання різних задач.

20.4.1. Кластерні системи

Кластером (cluster) називають розподілену систему, що складається із набору взаємозалежних комп'ютерів, які працюють разом як єдиний, інтегрований обчислювальний ресурс [18, 22, 60].

Окремі комп'ютери в рамках кластера називають вузлами (nodes). Вузли оснащені таким самим апаратним забезпеченням, що і автономний комп'ютер (пам'яттю, засобами введення-виведення, у більшості випадків – жорстким диском),

і можуть бути використані окремо поза кластером. Вони бувають однопроцесорними і багатопроцесорними, на кожному із них виконується окрема копія операційної системи. Усі вузли кластера загалом мають бути пов'язані високопродуктивною мережею (для цього можна скористатися такими технологіями, як Fast Ethernet, Gigabit Ethernet тощо).

Кластер має відображатися користувачам як єдина система, що володіє більшою продуктивністю, ніж окремі його вузли. Цю властивість називають властивістю єдиного образу системи (single system image).

Тут кластер розумітимемо як систему, що складається із невеликої кількості вузлів (від 2 до кількох сотень), розташованих поблизу один від одного (часто в межах однієї кімнати) і з'єднаних локальною мережею. Для позначення інтегрованих співтовариств комп'ютерів більшого масштабу, з'єднаних глобальними мережами, використовують термін grid-системи. Такі системи буде описано далі у розділі 20.4.2.

Кластерні технології дають змогу підвищувати обчислювальні можливості внаслідок використання стандартного і доступного за ціною програмного й апаратного забезпечення. Масштабування кластера (додавання в нього обчислювальних ресурсів) просте і не вимагає великих витрат: для додавання нового вузла у кластер достатньо підключити до відповідної мережі підходящий за характеристиками комп'ютер і запустити на ньому необхідне програмне забезпечення.

Програмні компоненти кластерних систем

Перелічимо основні програмні компоненти кластерних систем (рис. 20.3).

- На кожному вузлі кластера має виконуватись *ОС*, що підтримує багатопотоковість, мережні інтерфейси і протоколи (більшість кластерів реалізують зв'язок між вузлами на базі TCP/IP) та інші необхідні можливості. Особливості кластерних вирішень на основі Linux і Windows XP розглянемо далі.

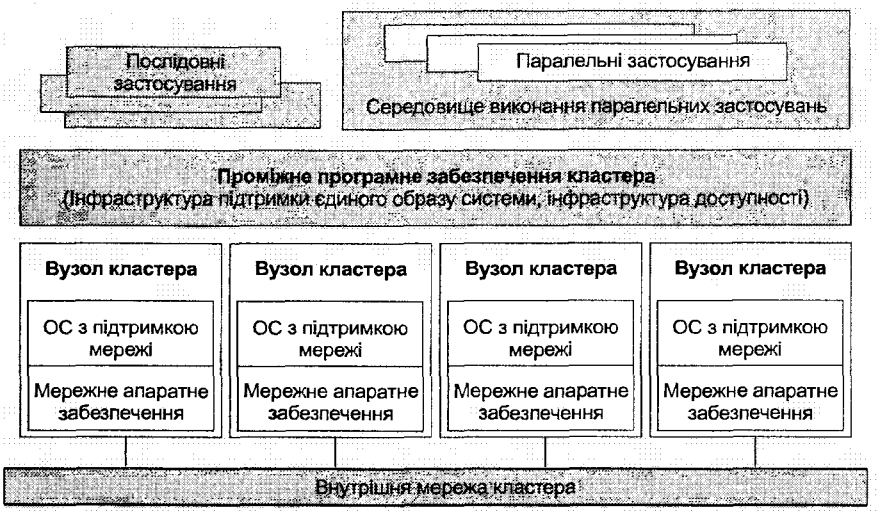

Рис. 20.3. Архітектура кластерної системи

- Функціонування кластера забезпечують проміжним програмним забезпеченням (ППО) кластера (cluster middleware).
- Обчислювальні ресурси кластера використовують паралельні застосування. Код таких застосувань розбитий на низку потоків або процесів, розрахованих на паралельне виконання на кількох вузлах кластера. Програмне забезпечення кластера має включати засоби підтримки розробки таких застосувань (компілятори, бібліотеки тощо).

Кластер також може виконувати код звичайних послідовних застосувань, але їхня продуктивність при цьому не буде вищою за продуктивність, досягнуту в разі виконання на одному процесорі (за законом Амдала).

Класифікація кластерних систем

Є різні підходи до класифікації кластерних систем [60, 101]. Насамперед можна виділити категорії таких систем, залежно від їхнього призначення.

- *Обчислювальні кластери* (High-Performance, HP) призначені для виконання паралельних програм, що описують вирішення складних наукових проблем (розшифрування генома, синтезу молекул тощо). Багато з них сьогодні потребують обчислювальних потужностей, яких неможливо досягти у разі використання окремих комп'ютерів. Фактично, такий кластер – це аналог багатопроцесорного суперкомп'ютера, але при цьому коштує значно дешевше. Більшу частину наукових обчислювальних задач у наш час розв'язують на таких кластерах.
- *Кластери з вирівнюванням навантаження* (load balancing) дають можливість розподіляти процесорне або мережне навантаження порівну між вузлами кластера. Вирівнювання навантаження потрібне, якщо велика кількість користувачів виконує один і той самий набір застосувань або створює багато мережних з'єднань одного типу (наприклад, у веб-серверах із високим вхідним навантаженням). Звичайно за розподіл навантаження відповідає окремий сервер, що виконується на одному із вузлів.
- *Кластери з резервуванням* (High Availability, HA) потрібні для забезпечення підвищеної надійності програмного та апаратного забезпечення комп'ютерних систем. Такі кластери надають можливість організовувати гаряче резервування окремих компонентів або вузлів системи. У разі виходу з ладу одного із вузлів обслуговування відразу продовжує інший вузол, при цьому для користувача робота системи не порушена.

На практиці часто можна зустріти кластери, які поєднують властивості різних категорій. Так, обчислювальні кластери можуть реалізовувати базові функції резервування і вирівнювання навантаження.

Ще один вид класифікації пов'язаний із контролем над вузлами кластера.

- *Виділений кластер* (dedicated) передбачає, що його вузли перебувають під єдиним контролем і можуть бути використані лише в рамках кластера.
- *Невиділений кластер* (non-dedicated) складається із вузлів, що належать різним власникам і переважно працюють незалежно. У цьому разі в рамках кластера використовують деяку частину обчислювальних ресурсів вузла (наприклад, процесорний час, не зайнятий виконанням власних застосувань).

Проміжне програмне забезпечення кластерних систем

ППО кластера звичайно складається із двох рівнів підтримки кластерних операцій.

- *Інфраструктура підтримки єдиного образу системи* (Single System Image Infrastructure, SSI) об'єднує разом операційні системи всіх вузлів для забезпечення уніфікованого доступу до системних ресурсів.
- *Інфраструктура доступності системи* (System Availability Infrastructure, SAI) забезпечує відновлення системи після збою окремих її вузлів .

Назвемо деякі характеристики кластерної системи, підтримку яких забезпечує ППО кластера.

- Єдина точка входу дає змогу користувачу з'єднуватися із кластером як з окремим комп'ютером, при цьому ППО має визначити, який із вузлів виконуватиме аутентифікацію.
- Єдина ієрархія файлової системи забезпечує те, що після входу у систему користувач сприймає файлову систему кластера як єдину ієрархію файлів і каталогів незалежно від їхнього фізичного місцезнаходження.
- Єдина віртуальна мережа дає змогу коду, що виконується на будь-якому вузлі кластера, отримувати доступ до зовнішніх мережних з'єднань, навіть якщо вони не підключені фізично до цього вузла.
- Єдиний адресний простір пам'яті об'єднує всі адресні простори окремих вузлів. Цей адресний простір доступний для всіх вузлів кластера.

  Інфраструктура доступності системи реалізує низку додаткових характеристик.
- Єдиний простір процесів дає змогу виконувати процеси однаково на будь-якому вузлі. Зокрема, всі процеси мають ідентифікатори, унікальні в межах кластера; результат виконання `fork()` може бути виконаний на іншому вузлі; дозволено взаємодію між процесами, які виконуються на різних вузлах.
- Підтримка контрольних точок (checkpoints) дає можливість періодично зберігати стан процесу під час його виконання. У разі виходу вузла з ладу процес може бути запущений на іншому вузлі з використанням збереженого стану – без втрати результатів обчислень. Дозволена також *міграція процесів* між вузлами під час їхнього виконання для вирівнювання навантаження.

Зазначимо, що загалом ППО кластера не зобов'язане виконуватись однаково на всіх його вузлах. Звичайно реалізація деякої функції вимагає виконання спеціального сервера на одному з вузлів (керуючому вузлі) і клієнтського коду на інших вузлах.

Засоби розробки паралельного програмного забезпечення

Оскільки зв'язок між вузлами кластера здійснюють зазвичай із використанням TCP/IP, взаємодія між компонентами паралельного програмного забезпечення, що виконуються на різних вузлах, може бути реалізована на основі інтерфейсу сокетів. Це, однак, є складним завданням, оскільки такий інтерфейс не містить жодних спеціалізованих засобів підтримки паралельних обчислень. У разі використання сокетів для кожної задачі потрібне розроблення необхідних структур даних, реалізація синхронізації, координації, обробки помилок тощо.

На практиці паралельні програми для кластерів розробляють із застосуванням засобів вищого рівня, що використовують сокети як базовий механізм. Найчастіше при цьому користуються інтерфейсом обміну повідомленнями, прикладами реалізації якого є PVM і MPI [6, 27, 57].

- *PVM* (Parallel Virtual Machine) – це мобільна бібліотека обміну повідомленнями і середовище виконання паралельного коду. Вона доступна для різних апаратних архітектур (багатопроцесорних систем, кластерів тощо) та різних ОС (зокрема Linux і Windows XP).
- Специфікація *MPI* (Message Passing Interface) розроблена для задання стандарту організації обміну повідомленнями у розподілених системах. Є реалізації MPI для різних ОС (зокрема для Linux і Windows XP доступний пакет MPICH).

Під час розробки паралельних застосувань широко застосовуються алгоритми, описані в розділі 7 (методи ведучого-веденого, портфеля задач і конвеєра). Зазначимо, що за умов розподіленої системи замість потоків у цих алгоритмах можна розглядати процеси, а замість примітивів синхронізації – примітиви обміну повідомленнями.

Кластерна архітектура Beowulf

Архітектура *Beowulf* [64] була розроблена в 1994 році як результат проекту, виконаного в Агентстві космічних досліджень США (NASA). Головною її особливістю є відмова від використання дорогого апаратного забезпечення, щоб досягти оптимального співвідношення ціни і продуктивності. Сьогодні вона є найрозповсюдженішою кластерною архітектурою.

Розглянемо характеристики Beowulf-кластера.

- Основною метою його використання є виконання паралельних обчислень (Beowulf-кластер завжди є обчислювальним кластером).
- Вузлами такого кластера є стандартні персональні комп'ютери з архітектурою IA-32.
- Усі вузли є виділеними і не можуть бути використані для інших цілей.
- Усе програмне забезпечення має бути доступне у вихідних кодах. Більшість Beowulf-кластерів працює під керуванням Linux.
- Зв'язок між вузлами здійснюють через виділену внутрішню мережу, яка використовує Ethernet, Fast Ethernet або інші мережні технології, підтримувані Linux. Зауважимо, що, як і для вузлів, основною вимогою до мережі є стандартність і низька вартість компонентів.

Звичайно один із вузлів є головним (head node). Його переважно оснащують монітором і клавіатурою, зв'язують із зовнішньою мережею, і він відіграє роль керуючого вузла для різних функцій системи. Рекомендують, щоб інші обчислювальні вузли (computing nodes) мали однакову апаратну конфігурацію (так легше підтримувати виконання паралельних застосувань).

Наведемо приклади ППО, яке можна використовувати у Beowulf-системах.

- Для підвищення продуктивності мережного обміну є можливість встановити набір модифікацій ядра Linux, що реалізують підтримку паралельного використання кількох мережних інтерфейсів (у цьому разі всі інтерфейси розглядають як один віртуальний канал із більшою пропускною здатністю).

- Загальний образ системи може бути реалізований із використанням пакета розподіленого простору процесів Beowulf (Beowulf distributed Process space, BPROC). Він складається із модифікацій ядра і утиліт режиму користувача, що підтримують єдиний простір процесів.

Під час розробки паралельного програмного забезпечення для Beowulf-систем можна використовувати будь-які засоби, доступні в Linux (наприклад, PVM або реалізації стандарту MPI).

Кластерна архітектура Windows XP

Серверні версії Windows XP (зокрема Windows Server 2003) містять засоби реалізації кластерів із резервуванням (серверні кластери) і кластерів із вирівнюванням навантаження (служба мережного вирівнювання навантаження, Network Load Balancing). Далі зупинимося на серверних кластерах [102].

Основою архітектури серверних кластерів є кластерна служба (Cluster Service) — набір компонентів кожного вузла, що підтримують функціонування кластера. Компоненти кластерної служби працюють в адресному просторі окремого фонового процесу.

Програмні та апаратні компоненти вузлів, якими керує кластерна служба, називають ресурсами. До ресурсів належать апаратні пристрої (диски, мережні інтерфейси) і логічні об'єкти (IP-адреси, застосування, бази даних). Ресурси можуть перебувати у підключеному і відключеному стані. Розрізняють локальні ресурси вузлів і спільні ресурси, доступні для всіх вузлів кластера (спільний масив дисків, спільна мережа). Спеціальний ресурс, який називають ресурсом кворуму (quorum resource), використовують для зберігання службової інформації, спільної для всього кластера. Його реалізують на базі одного із дисків спільного масиву.

Ресурси об'єднані у групи ресурсів. Така група звичайно складається із логічно пов'язаних ресурсів (наприклад, застосування і його даних). Власником групи у конкретний момент може бути тільки один вузол кластера. Для кожної групи задано, на якому вузлі їй переважно потрібно виконуватися і куди слід її перевести у разі виходу поточного вузла з ладу.

Найважливіші компоненти кластерної служби такі.

- *Менеджер вузлів* (Node manager) функціонує на кожному вузлі і підтримує локальний список поточних вузлів кластера. Для цього він періодично надсилає тактові повідомлення (heartbeats) аналогічним менеджерам інших вузлів. Коли один із них не відповідає на повідомлення, відповідний вузол вважають таким, що вийшов із ладу; про це негайно повідомляють інші вузли, внаслідок чого вони обновлюють свої списки поточних вузлів (стається подія перегруповання — regroup event).
- *Менеджер бази даних* (Database manager) керує конфігураційною базою даних кластера, яка містить інформацію про всі фізичні та логічні елементи кластера (вузли, типи і групи ресурсів, конкретні ресурси) і зберігається у реєстрі кожного вузла і в ресурсі кворуму. Цю інформацію використовують для відстеження поточного і бажаного стану кластера.
- *Менеджер виведення з ладу* (Failover manager) відповідає за обробку ситуації, коли один із вузлів виходить із ладу (failover) або повертається до роботи

(failback). Вихід із ладу може статися випадково (після апаратного або програмного збою) або навмисно (з ініціативи адміністратора кластера). При цьому групи ресурсів перерозподіляються між іншими вузлами з урахуванням поточного навантаження на вузли і переваг, заданих для цих груп. У разі поверненні вузла до роботи менеджер виведення з ладу переводить групи ресурсів назад під керування цього вузла. Цей менеджер, крім того, може зупиняти і перезапускати окремі ресурси у разі їхніх збоїв.

20.4.2. Grid-системи

Останнім часом розвиток технологій розподілених систем привів до ситуації, коли стало можливим організовувати системи, що дають змогу використати вільні обчислювальні ресурси, розподілені по всьому світу. Розглянемо особливості таких grid-систем [48, 53, 72].

Обчислювальні grid-системи дають можливість спільно використовувати географічно та організаційно розподілені неоднорідні обчислювальні ресурси, відображаючи їх як єдиний, уніфікований обчислювальний ресурс. До ресурсів, які можна таким чином використовувати, належать окремі комп'ютери, кластери, засоби зберігання даних, мережні ресурси тощо.

Назва «grid» відображає аналогію між використанням такого уніфікованого ресурсу і доступом до електричної мережі (power grid). Користувачі електромережі отримують доступ до ресурсу (електроенергії) незалежно від конфігурації джерел енергії та ліній електропередачі. Аналогічно для користувачів grid-системи не має значення, який із компонентів надав їм обчислювальний ресурс.

Grid-системи містять такі компоненти:

- засоби, що транслюють запити користувачів у запити до ресурсів grid-системи (комп'ютерів, мереж, дискового простору, баз даних тощо);
- засоби, які виконують пошук ресурсів, їхній підбір і розміщення, планування і координацію обчислювальних задач, а також збирання результатів;
- засоби безпеки, які дають змогу керувати аутентифікацією та авторизацією користувачів. Вони мають підтримувати єдиний вхід у систему, відображення на механізми захисту локальних систем, можливість запуску застосувань із правами користувача;
- засоби розробки застосувань, що використовують особливості grid-архітектури.

Основним призначенням grid-систем є підтримка розв'язування задач, що вимагають великих обчислювальних ресурсів. Цим вони подібні до обчислювальних кластерів. Але ці технології мають істотні відмінності [66].

- Grid-системи є географічно розподіленими (пов'язаними глобальними мережами) і неоднорідними (у них входять компоненти із різною апаратною і програмною архітектурою). Крім того, їхні компоненти можуть перебувати в різному адміністративному підпорядкуванні. Вузли кластерів зазвичай пов'язані локальною мережею, перебувають у централізованому підпорядкуванні та мають однакову архітектуру.
- Кластерна архітектура звичайно містить у собі централізований менеджер ресурсів. Для grid-архітектур внаслідок більшого масштабу системи кожний вузол має свій менеджер ресурсів.

- Вузли grid-систем завжди є невиділеними. Будь-який обчислювальний ресурс, що входить у таку систему, може водночас використовуватися для інших цілей. Для використання в межах grid-системи виділяються тільки ресурси, вільні в конкретний момент.
- Конфігурація grid-систем постійно змінюється. Хоча кластери розраховані на те, що їхня конфігурація може змінитися, такі зміни для них не є звичайною ситуацією.
- Grid-системи можуть розподіляти найрізноманітніші ресурси. Крім обчислювальних потужностей, до них належать мережні ресурси, ресурси зберігання даних, інформаційні ресурси (доступ до розподілених баз даних), програмні продукти і апаратні пристрої.

Оскільки grid-системи є неоднорідними, велике значення у їхній розробці відіграє стандартизація. Організовано спеціальний комітет зі стандартизації grid-розробок — Grid Forum, результатом роботи якого є загальноприйнятий стандарт Open Grid Service Architecture (OGSA). Існує програмна реалізація цього стандарту — Globus Toolkit, яку можна використати для практичної розробки grid-систем.

Є думка, що в найближчому майбутньому Інтернет буде перетворено у глобальну мережу, організовану відповідно до grid-архітектури.

Висновки

- Організація паралельного виконання коду на кількох процесорах — ефективний засіб підвищення продуктивності комп'ютерних систем. Основними архітектурами, що підтримують таке виконання, є багатопроцесорні та розподілені системи.
- Основним підходом до організації багатопроцесорних ОС є симетрична багатопроцесорність (SMP), за якої одна копія ядра ОС виконується на всіх процесорах системи. Для організації підтримки SMP у ядрі ОС виділяють критичні ділянки, кожна з яких може бути виконана тільки одним процесором у конкретний момент часу.
- Планування у багатопроцесорних системах відрізняється тим, що потрібно визначати не лише потік, який має перейти до виконання, але й процесор, на якому цей потік має виконуватися.
- Розподілені системи відрізняються від багатопроцесорних тим, що в них процесори перебувають у складі окремих комп'ютерів, з'єднаних мережею. Основними технологіями розробки застосувань для таких систем є передавання повідомлень і віддалені виклики процедур.
- Важливим прикладом реалізації розподілених систем є розподілені файлові системи. Вони дають можливість прозоро для користувачів організувати роботу із віддаленими файлами.
- Сучасні архітектури розподілених систем (кластерні та grid-архітектури) дають змогу об'єднувати обчислювальні потужності багатьох комп'ютерних систем для розв'язування задач, які неможливо розв'язати на окремих комп'ютерах.

Контрольні запитання та завдання

1. Перелічіть переваги і недоліки використання кеша в багатопроцесорних системах.
2. Чому реалізація критичної секції через заборону переривань не може бути використана в багатопроцесорних системах?
3. Назвіть причини, через які планувальники в багатопроцесорних системах не використовують спільну для всіх процесорів чергу готових потоків?
4. У чому полягає потенційна небезпека передачі покажчиків як параметрів у віддалені виклики процедур? Які програмні вирішення можуть бути запропоновані в цьому випадку?
5. Які проблеми вирішує автоматична генерація заглушок для RPC?
6. Розробіть код віддаленої процедури на основі Sun RPC і Microsoft RPC, що приймає як параметр шлях до файла на віддаленій системі і повертає перші 100 байт цього файла. Розробіть клієнт-серверну систему, подібну до створеної у завданні 11 з розділу 11, з використанням цієї процедури. При цьому клієнт повинен формувати і виконувати виклик віддаленої процедури з використанням отриманого від користувача імені файла та інформації про сервер, одержувати результат і відображати його на стандартний вивід. Сервер повинен виконувати віддалену процедуру і повертати результат.
7. Скільки повідомлень має бути відправлено внаслідок виконання протоколу 2PC за наявності одного координатора і N учасників?
8. Запропоновано таку модифікацію протоколу 2PC. На першому етапі координатор надсилає повідомлення M_{COMMIT} всім учасникам, потім очікує відповіді від кожного з них. Якщо хоча б один із них відішле повідомлення M_{ABORT}, координатор надсилає всім учасникам M_{GLOBAL_ABORT}, інакше додаткових повідомлень надіслано не буде. У чому недолік такого алгоритму?
9. У яких випадках кешування даних на клієнті може підвищити продуктивність розподіленої файлової системи, а в яких — знизити?
10. Поясніть, чому для розподілених файлових систем зменшення розміру блоку спричиняє зменшення навантаження на мережу, якщо використовують кешування даних на клієнті?
11. Які з перерахованих операцій є ідемпотентними:
    а) записування блоку даних у файл;
    б) запит списку користувачів, що працюють у цей момент на віддаленій системі;
    в) поміщення грошей на банківський рахунок?
12. Процес на клієнтському комп'ютері викликав операцію читання з файла, що міститься на NFS-розділі. Опишіть етапи виконання цієї операції.

Література та посилання

1. *Алексеев Д., Видревич Е., Волков А.* и др. Практика работы с QNX. — М.: КомБук, 2004. — 432 с.
2. *Альбитц П., Ли К.* DNS и BIND. — 4-е изд. — М.: Символ-Плюс, 2004. — 696 с.
3. *Беляков М. И., Рабовер Ю. И., Фридман А. Л.* Мобильная операционная система. — М.: Радио и связь, 1991. — 208 с.
4. *Бэкон Дж., Харрис Т.* Операционные системы. — К.: Издат. группа BHV; СПб.: Питер, 2004. — 800 с.
5. *Вахалия Ю.* UNIX изнутри. — СПб.: Питер, 2003. — 844 с.
6. *Воеводин В. В., Воеводин Вл. В.* Параллельные вычисления. — СПб.: БХВ-Петербург, 2004. — 608 с.
7. *Гордеев А. В., Молчанов А. Ю.* Системное программное обеспечение. — СПб.: Питер, 2001. — 736 с.
8. *Дейтел Г.* Введение в операционные системы. — М.: Мир, 1987. — Т. 1. — 359 с.; Т. 2. — 398 с.
9. *Джонс Э., Оланд Дж.* Программирование в сетях Microsoft Windows. — СПб.: Питер, 2001. — 608 с.
10. *Зима В., Молдовян А., Молдовян В.* Безопасность глобальных сетевых технологий. — 2-е изд. — СПб.: БХВ-Петербург, 2003. — 368 с.
11. *Зыль С.* Операционная система реального времени QNX. От теории к практике. — 2-е изд. — СПб.: БХВ-Петербург, 2004. — 192 с.
12. *Иртегов Д. В.* Введение в операционные системы. — СПб.: БХВ-Петербург, 2002. — 624 с.
13. *Камер Д.* Компьютерные сети и Internet. Разработка приложений для Internet. — М.: Вильямс, 2002. — 640 с.
14. *Кастер Х.* Основы Windows NT и NTFS. — М.: Русская Редакция, 1996. — 440 с.
15. *Кельтон В., Лоу А.* Имитационное моделирование. — 3-е изд. — К.: Издат. группа BHV; СПб.: Питер, 2004. — 847 с.
16. *Кокорева О.* Реестр MS Windows Server 2003. — СПб.: БХВ-Петербург, 2003. — 640 с.
17. *Корнійшко Т., Мельник А., Мельник В.* Алгоритми та процесори симетричного блокового шифрування. — Львів: БаК, 2003. — 168 с.
18. *Крюков В. А.* Разработка параллельных программ для вычислительных кластеров и сетей. — М.: Ин-т прикладной математики им. М. В. Келдыша РАН, 2002. URL: http://www.keldysh.ru/dvm/dvmhtm1107/publishr/cldvm2002web.htm
19. *Кузнецов С. Д.* Операционная система UNIX. URL: http://www.citforum.ru/operating_systems/unix/contents.shtml
20. *Кузнецов С. Д.* Открытые системы, процессы стандартизации и профили стандартов. URL: http://www.citforum.ru/database/articles/art_19.shtml
21. *Куроуз Д., Росс К.* Компьютерные сети: многоуровневая архитектура Интернета. — 2-е изд. — СПб.: Питер, 2004. — 765 с.
22. *Лацис А.* Как построить и использовать суперкомпьютер. — М.: Бестселлер, 2003. — 274 с.

23. *Масленников М.* Практическая криптография. — СПб.: БХВ-Петербург, 2003. — 464 с.

24. *Митчелл М., Оулдем Д., Самьюэл А.* Программирование для Linux. Профессиональный подход. — М.: Вильямс, 2002. — 288 с.

25. *Моли Б.* UNIX/Linux. Теория и практика программирования. — М.: Кудиц-Образ, 2004. — 576 с.

26. *Немнюгин С. А., Комолкин А. В., Чаунин М. П.* Эффективная работа: UNIX. — СПб.: Питер, 2001. — 688 с.

27. *Немнюгин С. А., Стесик О. Л.* Параллельное программирование для многопроцессорных вычислительных систем. — СПб.: БХВ-Петербург, 2002. — 400 с.

28. *Олифер В. Г., Олифер Н. А.* Компьютерные сети. Принципы, технологии, протоколы. — 2-е изд. — СПб.: Питер, 2003. — 864 с.

29. *Олифер В. Г., Олифер Н. А.* Сетевые операционные системы. — СПб.: Питер, 2001. — 544 с.

30. *Померанц О.* Ядро Linux. Программирование модулей. — М.: Кудиц-Образ, 2000. — 112 с.

31. *Рихтер Д.* Windows для профессионалов: Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. — М.: Русская Редакция, 2001. — 752 с.

32. *Рихтер Д., Кларк Д.* Программирование серверных приложений для Microsoft Windows 2000. — СПб.: Питер; М.: Русская Редакция, 2001. — 592 с.

33. *Робачевский А.* Операционная система UNIX. — СПб.: БХВ-Петербург, 1999. — 528 с.

34. *Смит Р.* Аутентификация: от паролей до открытых ключей. — М.: Вильямс, 2002. — 432 с.

35. *Солдатов В. П.* Программирование драйверов для Windows. — М.: Бином, 2003. — 432 с.

36. *Сорокина С. И., Тихонов А. Ю., Щербаков А. Ю.* Программирование драйверов и систем безопасности. — СПб.: БХВ-Петербург; М.: Молгачева, 2002. — 256 с.

37. *Стивенс У. Р.* UNIX: взаимодействие процессов. — СПб.: Питер, 2001. — 576 с.

38. *Стивенс У. Р.* UNIX: разработка сетевых приложений. — СПб.: Питер, 2003. — 1088 с.

39. *Стивенс У. Р.* Протоколы TCP/IP. — СПб.: БХВ-Петербург, 2003. — 672 с.

40. *Столлингс В.* Криптография и защита сетей: принципы и практика. — 2-е изд. — М.: Вильямс, 2001. — 672 с.

41. *Столлингс В.* Операционные системы. — М.: Вильямс, 2002. — 848 с.

42. *Столлингс В.* Современные компьютерные сети. — 2-е изд. — СПб.: Питер, 2003. — 784 с.

43. *Таненбаум Э.* Компьютерные сети. — 4-е изд. — СПб.: Питер, 2004. — 992 с.

44. *Таненбаум Э.* Операционные системы. — СПб.: Питер, 2002. — 1040 с.

45. *Таненбаум Э., Ван Стеен М.* Распределенные системы. Принципы и парадигмы. — СПб.: Питер, 2003. — 880 с.

46. *Торвальдс Л., Даймонд Д.* Ради удовольствия. — М.: ЭКСМО-Пресс, 2002. — 286 с.

47. *Уолтон Ш.* Создание сетевых приложений в среде Linux: Руководство разработчика. — М.: Вильямс, 2001. — 464 с.

48. *Фостер Я.* и др. Grid-службы для интеграции распределенных систем//Открытые системы. — 2003. — № 1. URL: http://www.osp.ru/os/2003/01/020.htm

49. *Фролов И. Б.* Карманные компьютеры на основе Windows CE и Palm OS. — М.: Майор, 2004. — 448 с.

50. *Харт Дж. В.* Системное программирование в среде Win32. — М.: Вильямс, 2001. — 464 с.

51. *Ховард М., Леви М., Вэймир Р.* Разработка защищенных Web-приложений на платформе Microsoft Windows 2000. — СПб.: Питер; М.: Русская Редакция, 2001. — 464 с.

52. *Чан Т.* Системное программирование на C++ для UNIX. — К.: Издат. группа BHV, 1999. — 592 с.

53. *Черняк Л.* Grid как будущее компьютинга//Открытые системы. — 2003. — № 1. URL: http://www.osp.ru/os/2003/01/016.htm

54. *Шмидт Д.* Программирование сетевых приложений на C++. — Т. 1. — М.: Бином Пресс, 2003. — 304 с.

55. *Шнайер Б.* Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. — М.: Триумф, 2002. — 816 с.

56. *Щербаков А. Ю., Домашев А. В.* Прикладная криптография: использование и синтез криптографических интерфейсов. — М.: Русская Редакция, 2003. — 416 с.

57. *Эндрюс Г.* Основы многопоточного, параллельного и распределенного программирования. — М.: Вильямс, 2003. — 512 с.

58. *Aivazian T.* Linux Kernel 2.4 Internals//Linux Documentation Project, 2002. URL: http://www.moses.uklinux.net/patches/lki.html

59. *Bach M. J.* The Design of the UNIX Operating System. — Englewood: Cliffs; NJ: Prentice-Hall, 1986. (Пер. с англ.: Бах М. Архитектура операционной системы UNIX). URL: http://www.lib.ru/BACH/

60. *Baker M., Buyya R.* Cluster Computing at a Glance//High Performance Cluster Computing/Ed. R. Buyya. — Vol. 1. URL: http://buyya.com/cluster/v1chap1.pdf

61. *Bonwick J.* The Slab Allocator: An Object-Caching Kernel Memory Allocator//Sun Microsystems, 1994. URL: http://www.nondot.org/sabre/os/files/MemManagement/SlabAllocator.pdf

62. *Bovet D. P, Cesati M.* Understanding the Linux Kernel. — Chap. 10: Process Scheduling. — S.l.:O'Reilly, 2000. URL: http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html

63. *Bowman I., Holt R., Brewster N.* Linux as a Case Study: Its Extracted Software Architecture, 1999. URL: http://plg.uwaterloo.ca/~itbowman/pub/icse99.ps

64. *Brown R.G.* Engineering a Beowulf-style Compute Cluster. — S.l.: Duke University, 2003. URL: http://www.phy.duke.edu/resources/computing/brahma/beowulf_book/

65. *Dahlin M.* Basic Threads Programming: Standards and Strategy, 2002. URL: http://www.cs.utexas.edu/users/dahlin/Classes/UGOS/reading/programming-with-threads.pdf

66. *Deshpande A.* Grid Computing Introduction. URL: http://www.geocities.com/asmya_mail/gridinfo.html

67. *Drepper U., Molnar I.* The Native POSIX Thread Library for Linux, 2003. URL: http://people.redhat.com/drepper/nptl-design.pdf

68. Embedded Linux Consortium. URL: http://www.embedded-linux.org

69. *Esposito D.* A Programmer's Perspective on NTFS 2000. — Pt 1: Stream and Hard Link//MSDN Magazine. — 2000. — N 3. URL: http://msdn.microsoft.com/library/en-us/dnfiles/html/ntfs5.asp

70. *Esposito D.* A Programmer's Perspective on NTFS 2000. — Pt 2: Encryption, Sparseness, and Reparse Points//MSDN Magazine. — 2000. — N 5. URL: http://msdn.microsoft.com/library/en-us/dnfiles/html/ntfs2.asp

71. Executable and Linkable Format (ELF).
URL: http://www.nondot.org/sabre/os/files/Executables/ELF.pdf

72. *Foster I.* The Grid: A New Infrastructure for 21st Century Science//Physics Today, 2002.
URL: http://www.physicstoday.org/vol-55/iss-2/p42.html

73. *Franke H., Russell R., Kirkwood M.* Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux//Proceedings of the Ottawa Linux Symposium. — Ottawa, ON, 2002. — P. 479–495.
URL: ftp://ftp.stearns.org/pub/doc/ols2002_proceedings.pdf.gz

74. *Gorman M.* Understanding the Linux Virtual Memory Manager, 2003.
URL: http://www.skynet.ie/~mel/projects/vm/guide/pdf/understand.pdf

75. *Hall B.* Beej's Guide to Network Programming, 2001.
URL: http://www.ecst.csuchico.edu/~beej/guide/ net/

76. *Kalinsky D., Barr M.* Introduction to Priority Inversion. Embedded.com, 2002.
URL: http://www.embedded.com/story/OEG20020321S0023

77. *Kegel D.* The C10K Problem, 2004. URL: http://www.kegel.com/c10k.html

78. *Lampson B.* Practical Concurrency//MIT, 1999.
URL: http://web.mit.edu/6.826/archive/S99/14.pdf

79. *Love R.* Kernel Korner: CPU Affinity//The Linux Journal, 2003.
URL: http://www.linuxjournal.com/article.php?sid=6799

80. *Manrique D.* X Window System Architecture Overview HOWTO//Linux Documentation Project, 2001.
URL: http://www.tldp.org/HOWTO/XWindow-Overview-HOWTO

81. *McKusick M. K., Bostic K., Karels M. J., Quarterman J. S.* The Design and Implementation of the 4.4BSD Operating System. — S.l.:Addison-Wesley, 1996.
URL: http://www.freebsd.org/doc/en_US.ISO8859-1/books/design-44bsd/book.html

82. *McKusick M. K., Joy W., Leffler S., Fabry R.* A Fast File System for UNIX. — Berkeley: University of California, 1984.
URL: http://docs.freebsd.org/44doc/smm/05.fastfs/paper.html

83. *Molnar I.* Ultra-scalable O(1) SMP and UP scheduler//Linux Weekly News, 2002.
URL: http://lwn.net/2002/0110/a/scheduler.php3

84. Multithreaded Programming Guide//Sun Microsystems, 2002.
URL: http://docs.sun.com/db/doc/805-5080

85. Newsgroup comp.os.research Frequently Asked Questions.
URL: http://www.faqs.org/faqs/by-newsgroup/comp/comp.os.research.html

86. Newsgroup comp.programming.threads Frequently Asked Questions.
URL: http://www.lambdacs.com/cpt/FAQ.html

87. *Nordell M.* Windows 2000 Junction Points//The Code Project, 2002.
URL: http://www.codeproject.com/w2k/junctionpoints.asp

88. Operating Systems Resource Center. URL: http://www.nondot.org/sabre/os/articles.htm

89. *Pietrek M.* An In-Depth Look into the Win32 Portable Executable File Format//MSDN Magazine. — 2002. — N 2–3.
URL: http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx

90. *Pranevich J.* The Wonderful World of Linux 2.6.
URL: http://www.kniggit.net/wwol26.html

91. RSA Laboratories' Frequently Asked Questions About Today's Cryptography.
RSA Security Inc., 2000. URL: http://www.rsa.com/rsalabs/newfaq

92. *Rinard M.* Operating Systems Lecture Notes.
URL: http://www.cag.lcs.mit.edu/~rinard/osnotes/

93. *Rodriguez A.* et al. TCP/IP Tutorial and Technical Overview. IBM, 2001.
URL: http://www.ibm.com/redbooks

94. *Rubini A., Corbet J.* Linux Device Drivers, 2nd edition. O'Reilly, 2001.
URL: http://www.xml.com/ldd/chapter/book/

95. *Rusling D.* The Linux Kernel//Linux Documentation Project, 1999.
URL: http://www.linuxdoc.org/LDP/tlk/tlk.html

96. *Russinovich M.* Inside I/O Completion Ports, 1998.
URL: http://www.sysinternals.com/ntw2k/info/comport.shtml

97. *Russinovich M.* Inside the Windows NT Scheduler. – Pt 1//Windows Magazine, 1997.
URL: http://www.winnetmag.com/Articles/Index.cfm?ArticleID=302

98. *Russinovich M.* Inside the Windows NT Scheduler. – Pt 2//Windows Magazine, 1997.
URL: http://www.winnetmag.com/Articles/Index.cfm?ArticleID=303

99. *Russinovich M.* Windows NT Security. Pt 1–2.
URL: http://www.secinf.net/windows_security/Windows_NT_Security_Part_1[2].html

100. *Schmidt D., Pyarali I.* Strategies for Implementing POSIX Condition Variables on Win32. – Washington: University St. Louis, MI, 1999.
URL: http://www.cs.wustl.edu/~schmidt/win32-cv-1.html

101. *Schneider N.* What is Clustering?, 2003.
URL: http://www.linuxgeek.net/index.pl/what_is_clustering

102. Server Clusters: Architecture Overview. Microsoft Corporation, 2003.
URL: http://www.microsoft.com/windowsserver2003/techinfo/overview/servercluster.mspx

103. The Open Group Base Specifications Release 6. The IEEE and The Open Group, 2003.
URL: http://www.opengroup.org

104. *Thompson K., Ritchie D. M.* The UNIX Time-Sharing System//Communications of the ACM. – 1974. – Vol. 17, N 7. – P. 365–375. (переработанная версия доступна по URL http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html)

105. Unix Programming FAQ. URL: http://www.erlenstar.demon.co.uk/unix/faq_toc.html

106. Unix Socket FAQ. URL: http://www.developerweb.net/sock-faq/

107. *Wheeler D. A.* Program Library HOWTO//Linux Documentation Project, 2002.
URL: http://www.dwheeler.com/program-library

108. *Wheeler D. A.* Secure Programming for Linux and Unix HOWTO.
URL: http://www.dwheeler.com/secure-programs

109. *Wilson P., Johnstone M., Neely M., Boles D.* Dynamic Storage Allocation: A Survey and Critical Review. URL: ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps

110. Winsock Programmer's FAQ. URL: http://tangentsoft.net/wskfaq/

Алфавітний покажчик

D

E

M

N

O

P

Q

R

S

К

P

С