

**І. М. Федотова-Півень,
О. Б. Півень**

ОПЕРАЦІЙНІ СИСТЕМИ



УНІВЕРСИТЕТ ДЕРЖАВНОЇ ФІСКАЛЬНОЇ СЛУЖБИ УКРАЇНИ

І. М. Федотова-Півень,
О. Б. Півень

ОПЕРАЦІЙНІ СИСТЕМИ

Навчальний посібник

Ірпінь
2021

УДК 004.451
ББК 32.973-018.2я7
Ф34

*Рекомендовано до друку вченою радою
Університету державної фіскальної служби України
(протокол № 14 від 23 листопада 2021 року)*

Рецензенти:

Горбовий Артур Юліанович, доктор технічних наук, професор, професор кафедри інтелектуальних управляючих та обчислювальних систем, директор ННІ інформаційних технологій Університету державної фіскальної служби України;

Бабенко Віра Григорівна, доктор технічних наук, доцент, професор кафедри інформаційної безпеки та комп'ютерної інженерії Черкаського державного технологічного університету.

Федотова-Півень І. М.

Ф34

Операційні системи : навчальний посібник /

І. М. Федотова-Півень, О. Б. Півень. – Ірпінь : Університет державної фіскальної служби України, 2021. – 286 с.

ISBN 978-966-337-662-2

У навчальному посібнику висвітлені основні питання щодо сучасного стану операційних систем, їх основних складових, архітектури, принципів і концепцій побудови. Містить теоретичні та практичні матеріали, матеріали для самостійної роботи студентів.

Може бути використаний у навчальному процесі підготовки здобувачів вищої освіти галузі знань 12 Інформаційні технології.

УДК 004.451
ББК 32.973-018.2я7

ISBN 978-966-337-662-2

© Федотова-Півень І. М.,
Півень О. Б., 2021
© Університет ДФС України, 2021

ЗМІСТ

СПИСОК СКОРОЧЕНЬ	8
------------------------	---

ВСТУП	10
-------------	----

ЧАСТИНА I

РОЗДІЛ 1. АРХІТЕКТУРА, ПРИЗНАЧЕННЯ І ФУНКЦІЇ ОПЕРАЦІЙНИХ СИСТЕМ.....	12
1.1. Функції, класифікації і складові ОС	12
1.2. Структура операційних систем	17
1.3. Системні виклики	18
1.4. Поняття операційного і програмного середовища.....	19
1.5. Архітектура операційних систем.....	20
<i>1.5.1. Архітектура ОС Windows</i>	<i>30</i>
<i>1.5.2. Особливості сучасних архітектур ОС UNIX і Linux.....</i>	<i>33</i>
1.6. Операційна система Android.....	36
<i>1.6.1. Архітектура програмного забезпечення Android</i>	<i>37</i>
1.7. Архітектура iOS	39
1.8. Основні етапи розвитку операційних систем	41
Контрольні питання	44

РОЗДІЛ 2. ПРОЦЕСИ. ПОТОКИ. СИГНАЛИ.....	46
2.1. Основні відомості про процеси.	
Управління процесами	46
<i>2.1.1. Властивості та класифікація процесів.....</i>	<i>46</i>
<i>2.1.2. Моделі процесу.....</i>	<i>49</i>
<i>2.1.3. Опис процесів</i>	<i>52</i>
<i>2.1.4. Управління процесами.....</i>	<i>54</i>
<i>2.1.5. Стратегії планування і диспетчеризація процесів</i>	<i>57</i>
<i>2.1.6. Алгоритми в диспетчеризації з витісненням та без витіснення.....</i>	<i>61</i>
2.2. Взаємодія процесів	63

2.2.1. Проблема взаємного виключення і способи її вирішення	63
2.2.2. Апаратні способи досягнення взаємного виключення	67
2.2.3. Програмні способи досягнення взаємного виключення	68
2.3. Примітиви синхронізації	71
2.3.1. Семафори	71
2.3.2. М'ютекси	73
2.3.3. Монітори	73
2.4. Взаємні блокування	75
2.5. Класичні проблеми міжпроцесної взаємодії	84
2.6. Потоки	87
2.6.1. Основні операції з потоками та способи реалізації потоків	89
2.6.2. Організація процесів і потоків в Linux	91
2.7. Сигнали ОС UNIX	93
Контрольні питання	97

РОЗДІЛ 3. УПРАВЛІННЯ ПАМ'ЯТТЮ

В ОПЕРАЦІЙНИХ СИСТЕМАХ	100
3.1. Технології розподілу пам'яті	104
3.1.1. Фіксований розподіл пам'яті	105
3.1.2. Динамічний розподіл	106
3.1.3. Система двійників	106
3.1.4. Проста сторінкова організація	107
3.1.5. Проста сегментна організація	110
3.2. Віртуальна пам'ять	111
3.2.1. Передумови для організації віртуальної пам'яті	111
3.2.2. Сторінкова організація віртуальної пам'яті	116
3.2.3. Сегментація	117
3.2.4. Комбінація сегментації і сторінкової організації	118
3.2.5. Алгоритми управління віртуальною пам'яттю	118
3.2.6. Управління пам'яттю в сучасних версіях Windows	121
Контрольні питання	122

РОЗДІЛ 4. СИСТЕМА ВВЕДЕННЯ / ВИВЕДЕННЯ. ФАЙЛОВА СИСТЕМА	124
4.1. Система введення / виведення	124
<i>4.1.1. Програмні рівні введення / виведення</i>	<i>125</i>
4.2. Завдання файлової системи. Поняття файла	128
<i>4.2.1. Фізична організація файлової системи.....</i>	<i>128</i>
<i>4.2.2. Будова диска.....</i>	<i>131</i>
<i>4.2.3. Фізична організація та адресація файла.....</i>	<i>135</i>
4.3. Логічна організація файлової системи.....	139
<i>4.3.1. Типи файлів.....</i>	<i>139</i>
<i>4.3.2. Ієрархічна структура файлової системи.....</i>	<i>140</i>
<i>4.3.3. Атрибути файлів.....</i>	<i>142</i>
Контрольні питання	143
 РОЗДІЛ 5. ТЕНДЕНЦІЇ РОЗВИТКУ СУЧАСНИХ ОПЕРАЦІЙНИХ СИСТЕМ	 145
5.1. Сучасні тенденції в розвитку ОС	145
<i>5.1.1. Операційні системи різних великих фірм</i>	<i>146</i>
<i>5.1.2. Переваги 64-розрядної версії Windows</i>	<i>148</i>
<i>5.1.3. Windows 8 як новий етап у розвитку операційних систем Microsoft</i>	<i>154</i>
<i>5.1.4. Windows 10.....</i>	<i>155</i>
<i>5.1.5. ОС для хмарних обчислень (cloud computing). Windows Azure</i>	<i>157</i>
5.2. Особливості ОС для мобільних пристроїв	161
<i>5.2.1. Короткий огляд ОС для мобільних пристроїв</i>	<i>162</i>
<i>5.2.2. Розквіт ОС у 2000-х роках</i>	<i>165</i>
Контрольні питання	167

ЧАСТИНА II

РОЗДІЛ 6. ОПЕРАЦІЙНІ СИСТЕМИ LINUX ТА WINDOWS	170
6.1. Робота з файлами з командного рядка LINUX.....	170

6.2. Мова програмування Perl у Linux для Perl-скриптів	175
6.3. Створення, відслідковування і знищення процесів в ОС Linux	191
6.3.1. Моніторинг процесів.....	195
6.3.2. Потoki, програмні канали перенаправлення в ОС Linux	197
6.4. Потoki і процеси в ОС Windows	200
6.4.1. Створення і керування процесом в ОС Windows.....	200
6.4.2. Програмна реалізація потоків в ОС Windows	202
Контрольні питання	205

РОЗДІЛ 7. СУЧАСНИЙ x86ASSEMBLER.

АРХІТЕКТУРА ЯДРА x86-64.....	207
7.1. Сучасний x86Assembler. Архітектура ядра x86-64.....	207
7.1.1. Внутрішня архітектура процесора x86-64	207
7.1.2. Регістри загального призначення	208
7.1.3. Інструкції, операнди.....	209
7.1.4. Режими адресації пам'яті	209
7.1.5. Відмінності між архітектурами x86-64 та x86-32.....	210
7.2. Огляд набору інструкцій (команд) процесора x86-64	211
7.2.1. Використання основних інструкцій процесора x86-64	211
7.2.2. Некоректні інструкції	212
7.2.3. Нові інструкції процесора x86-64	212
7.3. Програмування ядра x86-64	214
7.3.1. Основи програмування x86-64	214
7.3.2. Арифметика цілих чисел.	215
7.3.3. Адресація пам'яті	215
7.3.4. Робота з цілими числами.....	218
7.3.5. Арифметика з плаваючою крапкою.	221
7.3.6. Способи виклику підпрограм у x86-64. Робота зі стеком	223
7.3.7. Використання необов'язкових регістрів загального призначення.	224
7.3.8. Робота з макросами.....	225

7.3.9. Робота з масивами і рядками у x86-64 асемблері	226
7.4. SIMD розширення архітектури x86-64	230
7.4.1. x86 – сумісні процесори: (SSE, SSE2, SSE3, AVX, AV2).....	230
7.4.2. Виконавче оточення (середовище) x86-AVX.....	230
7.5. Програмування x86-64 SIMD	231
7.5.1. Програмування x86-SSE-64. Робота із зображенням	231
7.5.2. Програмування x86-SSE-64. Робота перетворення зображень	235
7.5.3. Векторні масиви	240
7.5.4. Програмування x86-AVX-64.....	243
7.5.5. Робота з RGB зображенням	246
7.5.6. Інвертування матриці з використанням SSE-інструкцій.	251
7.6. Робота з багатоядерними архітектурами.....	258
7.6.1. Мікроархітектура процесора	258
7.6.2. Огляд багатоядерних процесорів.....	259
7.6.3. Мікроархітектура функціонування каналів передачі інформації	260
7.6.4. Виконавчий модуль	262
7.7. Оптимізація коду асемблера	263
7.7.1. Основи оптимізації	263
7.7.2. Оптимізація обчислень з плаваючою крапкою.....	264
7.7.3. Оптимізація розгалуження програми.....	265
7.7.4. Оптимізація роботи з даними	266
7.7.5. Методика оптимізації SIMD-коду	266
Контрольні питання	267
ТЕСТОВІ ПИТАННЯ	269
ТЕМИ ДЛЯ САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ	275
СЛОВНИК ОСНОВНИХ ТЕРМІНІВ.....	276
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	284

СПИСОК СКОРОЧЕНЬ

ALPC – advanced local procedure call
AOSP – Android Open Source Project
API – Application Programming Interface
BR – boot record
BSD – Berkeley Software Distribution
CDMA – Code Division Multiple Access
CTSS – Compatible Time-Sharing System
DMA – Direct Memory Access
EDGE (EGPRS) – Enhanced Data rates for GSM Evolution
EV-DO (EVDO) – Evolution-Data Only/Evolution-Data Optimized
FAT – file allocation table
FCFS – first come first server
FIFO – first input first output
FMS – Fortran Monitor System
GNU – GNU's Not Unix
GPRS – General Packet Radio Service
GSM – Groupe Spécial Mobile
HAL – Hardware Abstraction Layer
JCL – Job control language
JME – Java Micro Edition
JSE – Java Standard Edition
LIFO – last input first output
MMU – Memory Management Unit
NTFS – New Technology
Rdir – root directory
ResSecs – reserved sector
RR – round robin
SJN – short jump next

SMP – Symmetric Multiprocessing

SRT – short remain time

SSL – Secure Socket Layer

Vx-Fs – Veritas File System

Wi-Fi – Wireless Fidelity

WiMAX – Worldwide Interoperability for Microwave Access

WRK – Windows Research Kernel

АЛП – арифметико-логічний пристрій

ЕОМ – електронна обчислювальна машина

КР – критичний ресурс

КС – критична секція

ОП – оперативна пам'ять

ОС – операційна система

ПВВ – пристрій для введення/виведення

ПЗ – програмне забезпечення

ПК – персональний ком'ютер

СУБД – системи управління базами даних

ФС – файлова система

ЦП – центральний процесор

ЧНП – черга нових процесів

ЧРП – черга робочих процесів

ВСТУП

Освоєння принципів і алгоритмів, покладених в основу розробки операційних систем, вивчення їх внутрішньої будови є важливою складовою базової комп'ютерної підготовки у закладах вищої освіти за спеціальністю 122 Комп'ютерні науки.

Основне призначення цього навчального посібника полягає у системному відображенні основних понять щодо складу, основного призначення і функцій сучасних операційних систем. Беручи до уваги те, що розвиток операційних систем не стоїть на місці і в сучасній іноземній літературі досить широко і глибоко представлені питання, що стосуються розвитку, складу, функцій і основних процесів в операційних системах (автори Е. Тоненбаум, В. Столлінгс та ін.), основною метою цього посібника є подання існуючої обширної інформації в стислому і доступному для швидкого сприйняття вигляді.

Зміст посібника структуровано на дві частини, додатки та словник основних термінів.

Перша частина посібника присвячена короткому теоретичному курсу з основних питань дисципліни «Операційні системи». Теоретичний матеріал поданий у стислому вигляді, але охоплює всі питання, що вивчаються в цьому курсі. Метою першої частини є допомога здобувачам вищої освіти впорядкувати знання й інформацію, що отримані під час лекційного курсу.

Друга частина посібника містить курс для практичного застосування під час вивчення дисципліни «Операційні системи», а саме надано інформацію про програмну реалізацію керування процесами і потоками в ОС Linux та ОС Windows. Також представлено можливості сучасного x86Assembler для архітектури ядра x86-64 у разі програмування під 64x розрядну ОС Windows.

У *додатках* посібника подано тестові завдання для перевірки рівня засвоєння курсу, теми реферативних робіт для самостійної роботи студентів.

ЧАСТИНА I

РОЗДІЛ 1¹

АРХІТЕКТУРА, ПРИЗНАЧЕННЯ І ФУНКЦІЇ ОПЕРАЦІЙНИХ СИСТЕМ

1.1. Функції, класифікації і складові ОС

Операційна система (ОС) – це програма, що керує роботою інших програм як прикладних, так і системних і є інтерфейсом між програмами і апаратним забезпеченням комп'ютера. Тому можемо стверджувати, що найголовнішими функціями, які виконує операційна система, є:

1. *Інтерфейс між користувачем і комп'ютером.*
2. *Керування ресурсами.*

Також виділяють й інші функції операційних систем:

- стандартизований доступ до периферійних пристроїв (пристрої введення / виведення);
- завантаження програм в оперативну пам'ять і їх виконання;
- керування оперативною пам'яттю (розподіл між процесами, організація віртуальної пам'яті);
- керування доступом до даних енергонезалежних носіїв (жорсткий диск, оптичні диски тощо), організованих у певній файловій системі;
- забезпечення користувацького інтерфейсу;
- паралельне або псевдопаралельне виконання задач (багатозадачність);
- розподіл ресурсів обчислювальної системи між процесами;
- організація надійних обчислень, що полягає у обмежуванні доступу до ресурсів. Тобто унеможливлення впливу процесу на перебіг інших процесів та на сам ресурс;

¹ Під час підготовки цього розділу були використані літературні джерела [1–4], [9–11], [12], [15], [17], [21], [23] та ін.

- взаємодія між процесами: обмін даними, синхронізація;
- захист самої системи, а також користувацьких даних і програм від дій користувача або програм;
- багатокористувацький режим роботи та розділення прав доступу (автентифікація, авторизація).

Пересічним користувачем комп'ютерна система сприймається як набір додатків (прикладних програм) і його не цікавлять подробиці апаратного забезпечення комп'ютерної системи. Додатки створюють програмісти на одній з мов програмування, і для спрощення програмування функцій управління комп'ютером розроблено набір системних програм (деякі з них називаються утилітами), які реалізують часто використовувані функції (роботу з файлами, управління пристроями введення / виведення тощо). Програмісти використовують ці засоби під час розробки власних програм. Зрозуміло, що найбільш важливою частиною системних програм є ОС, що безпосередньо взаємодіє з апаратурою. Отже, маємо сприймати ОС як посередника, що забезпечує і спрощує користувачеві (програмісту), а також програмним додаткам доступ до різних служб, зменшує залежність від конкретної реалізації апаратного забезпечення. Також ОС управляє ресурсами, розподіляючи їх між програмами або у часі (розподіл доступу до процесора, принтеру), або в просторі (використання оперативної пам'яті, дискового простору).

На рисунку 1.1 подана ієрархічна структура програмного забезпечення.

На рисунку 1.2 подано більш докладну схему ієрархічної структури програмно-апаратного забезпечення комп'ютера.

Нижній рівень містить різні пристрої комп'ютера, що складаються з мікросхем, провідників, джерел живлення та ін. Цей рівень можна розділити на підрівні, наприклад контролери пристроїв, а потім самі пристрої. Можливий розподіл і на більшу кількість рівнів.

Вище розташований мікроархітектурний рівень, на якому фізичні пристрої розглядаються як окремі функціональні одиниці.

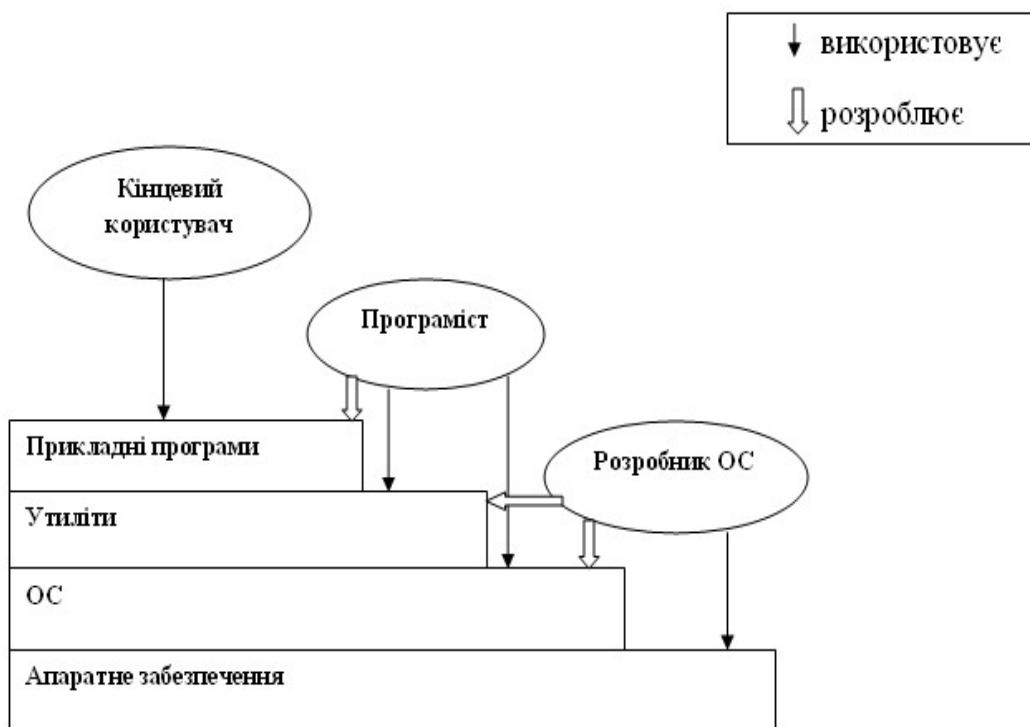


Рис. 1.1. Ієрархічна структура програмного забезпечення

На мікроархітектурному рівні містяться внутрішні реєстри центрального процесора (їх може бути декілька) й арифметико-логічні пристрої (АЛП) із засобами управління ними. На цьому рівні реалізується виконання машинних команд. У процесі виконання команд використовуються реєстри процесора і пристроїв, а також інші можливості апаратури.



Рис. 1.2. Ієрархічна структура програмно-апаратних засобів комп'ютера:

- рівні, доступні для розробників ОС;
- рівні, доступні для користувача і програміста

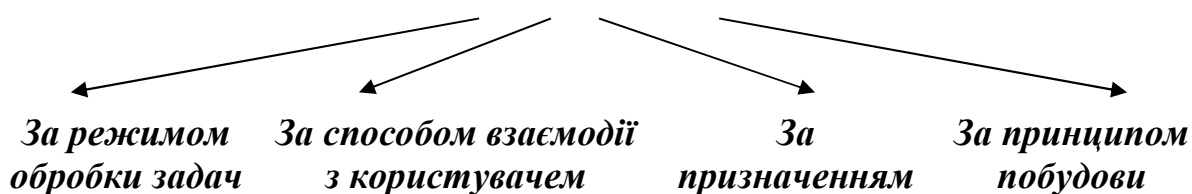
Команди, видимі для працюючого на асемблері програміста, формують рівень ISA (Instruction Set Architecture – архітектура системи команд), так звана машинна мова.

Операційна система призначена для того, щоб приховати всі ці складності. Отже, операційна система – це набір програм, які контролюють роботу прикладних програм і системних додатків і виконують роль інтерфейсу між користувачами, програмістами, прикладними програмами, системними додатками й апаратним забезпеченням комп'ютера.

Сервіси, що надають типові операційні системи:

- розробка програм;
- виконання програм;
- доступ до пристроїв введення / виведення;
- доступ до файлів;
- розподіл ресурсів;
- визначення й обробка помилок;
- облік використання ресурсів;
- засоби комунікації;
- захист даних і адміністрування.

Класифікація ОС



За режимом обробки задач:

- однопрограми (наприклад, MS-DOS);
- мультипрограми (більшість сучасних ОС).

За способом взаємодії з користувачем:

- діалогові ОС;
- системи пакетної обробки.

За призначенням:

- *ОС мейнфреймів* (орієнтовані на одночасну обробку великої кількості завдань, що потребують багато операцій введення/виведення; можуть виконувати три види обслуговування (часто всі разом): пакетну обробку, обробку транзакцій і роботу в режимі поділу часу. Приклад: OS/390, що створена на основі OS/360);

- *серверні ОС* (UNIX, Windows 2000, Linux);

- *мережеві ОС* (у разі роботи з такою ОС користувачі знають про існування в мережі безлічі ЕОМ, можуть реєструватися на віддалених комп'ютерах і копіювати файли з однієї ЕОМ на іншу. Кожен комп'ютер при цьому працює під управлінням локальної ОС і має свого власного локального користувача [користувачів]);

- *розподілені ОС* (розподілені ОС повинні мати в своєму складі спеціальні програми для оптимізації розпаралелювання задач. Приклади розподілених ОС: Mach, Chorus, Sprite);

- *ОС для персональних комп'ютерів* (Windows, Mac OS, Linux);

- *ОС реального часу* (обробляє завдання, що надходять упродовж заданих інтервалів часу, які не можна перевищувати; використовуються для керування технологічними процесами і лабораторними експериментами, в роботах систем управління повітряним рухом, у телекомунікаціях);

- *вбудовані ОС* (для забезпечення роботи мікрокомп'ютерів, вбудованих у різні побутові пристрої, мають такі самі характеристики, що й ОС реального часу, повинні займати невелике місце в пам'яті з урахуванням обмеження щодо обчислювальної потужності мікрокомп'ютерів). Приклади вбудованих ОС: PalmOS, Windows CE.

За принципом побудови:

- макроядерні (монолітні);
- мікроядерні;
- ієрархічні.

1.2. Структура операційних систем

Для більш докладного роз'яснення класифікації ОС за принципом побудови з'ясуємо, з чого складається операційна система.

До складу операційної системи входять:

- *ядро операційної системи*, що забезпечує розподіл та управління ресурсами обчислювальної системи;
- *базовий набір прикладного програмного забезпечення*;
- *системні бібліотеки та програми обслуговування*.

Ядро операційної системи – це набір функцій, структур даних та окремих програмних модулів, які завантажуються в пам'ять комп'ютера під час завантаження операційної системи та забезпечують три типи системних сервісів:

- управління введенням / виведенням інформації (підсистема – введення / виведення ОС);
- управління оперативною пам'яттю (підсистема управління оперативною пам'яттю ОС);
- управління процесами (підсистема управління процесами ОС).

Типова ОС складається з таких компонентів:

- керування процесами;
- керування пам'яттю;
- підсистема введення / виведення;
- файлова система;
- підтримка мережі;
- користувацький інтерфейс.

Структура ОС визначає, як пов'язані між собою її компоненти.

Монолітна структура (структура макроядра). Усі програми ОС монолітної структури зібрані в один модуль. Така ОС являє собою набір процедур, кожна з яких за необхідності може

викликати інші процедури, оскільки в монолітній ОС усі процедури відтрансльовані і скомпоновані в єдиний виконуваний файл. Порівняно з іншими структурами монолітну ОС важко розробляти, відлагоджувати, розширювати, і, як наслідок, у неї низька надійність, але більш висока швидкодія. Приклади таких ОС: MS-DOS, ранній Unix.

Ієрархічна структура (багатошарове ядро). Основні принципи побудови ОС ієрархічної структури:

- кожний рівень контролює свій ряд структур даних і надає функції доступу до них;
- кожний рівень нічого «не знає» про структуру рівнів, що перебувають нижче, про доступ до сервісів, який відбувається через певний інтерфейс;
- кожний рівень нічого «не знає» про існування рівнів, що перебувають вище.

Переваги ієрархічної ОС: модульність, простота відладки і підтримки, можливість заміни або модифікації окремого шару.

Структура мікроядра. У такій ОС ядро виконує невелику кількість більш важливих функцій: робота з адресним простором, міжпроцесна взаємодія, основне планування. Інші функції ОС виконуються процесами або серверами. Ці процеси запускаються в режимі користувача, а мікроядро працює з ними так само, як і з іншими додатками. При цьому спрощується розробка ОС, підвищується її гнучкість, але разом з тим відбувається зменшення швидкодії обчислювальних процесів.

1.3. Системні виклики

Функції ядра операційної системи можуть бути виконані внаслідок виконання в прикладних програмах спеціальних функцій – *системних викликів*.

Більшість ОС можуть працювати у двох режимах:

- у режимі користувача (*user mode*), в якому не дозволені привілейовані операції, прямий доступ до пристроїв введення / виведення і деяких керуючих регістрів;

– в *системному режимі (system mode)*, в якому дозволені привілейовані операції, прямий доступ до пристроїв введення / виведення і до всіх керуючих регістрів. Системний режим також називається режимом ядра, або режимом супервізора.

Необхідність системних викликів виникає тому, що прикладні програми не мають змоги самотужки визначити, за якими адресами перебувають функції ядра.

Системний виклик в один з машинно-залежних способів реалізує механізм отримання адрес функцій ядра та передачу в ці функції необхідних параметрів системного виклику, а також отримання результату системного виклику. Найчастіше системні виклики забезпечуються через систему переривань, завдяки чому адреса функції ядра не тільки обраховується апаратно (у процесі обробки переривання), але й забезпечується захист інформаційних ресурсів ядра. Системні виклики найчастіше мають синтаксис функції мови програмування, якою написано ядро ОС.

1.4. Поняття операційного і програмного середовища

Образно можна зазначити, що апаратура комп'ютера надає «сиру» обчислювальну потужність, а завдання операційної системи полягає в тому, щоб зробити використання цієї обчислювальної потужності доступним і за можливості зручним для користувача. Програміст може не знати деталі управління конкретними ресурсами (наприклад, диском) комп'ютера і повинен звертатися до операційної системи з відповідними викликами, щоб отримати від неї необхідні сервіси та функції. Цей набір сервісів і функцій і є операційним середовищем, в якому виконуються прикладні програми.

Отже, *операційне середовище* – це інтерфейси, що необхідні програмам і користувачам для отримання сервісів ОС. Операційне середовище включає в себе *програмне середовище* та інтерфейс користувача.

Програмне середовище, що безпосередньо утворене кодом ОС, називається основним. Крім нього, в ОС можуть бути організовані додаткові програмні середовища шляхом емуляції іншої

операційної системи. Отже, можливо забезпечити виконання програм, що створені для інших операційних систем.

Звернення користувачів (програм) до функцій ОС відбувається за допомогою систем запитів або викликів.

Для кожної ОС характерним є свій набір системних викликів, який являє собою сукупність системних викликів і правила їх застосування (*API*, Application Programming Interface – *інтерфейс прикладного програмування*).

Програма, що створена для однієї ОС, найімовірніше не буде працювати в іншій, оскільки ці ОС мають різні API. Для подолання такого обмеження розробляються програмні середовища.

Програмне середовище – це системне програмне оточення, що дозволяє виконувати системні запити від прикладних програм.

1.5. Архітектура операційних систем

Під архітектурою операційної системи розуміють структурну і функціональну організацію ОС на основі деякої сукупності програмних модулів. До складу ОС входять виконувані та об'єктні модулі стандартних для цієї ОС форматів, програмні модулі спеціального формату (наприклад, завантажувач ОС, драйвери введення / виведення), конфігураційні файли, файли документації, модулі довідкової системи тощо.

На архітектуру ранніх операційних систем зверталось мало уваги: по-перше, ні в кого не було досвіду в розробці великих програмних систем, а по-друге, проблема взаємозалежності та взаємодії модулів недооцінювалася. У монолітних ОС майже всі процедури могли викликати одна іншу. Така відсутність структури стала несумісною з розширенням операційних систем. Перша версія ОС OS/360 була створена колективом з 5 000 робітників за 5 років і містила понад 1 млн рядків коду. Розроблена дещо пізніше операційна система Mastsics містила до 1975 року вже 20 млн рядків. Стало зрозумілим, що розробка таких систем повинна вестися на основі модульного програмування.

Більшість сучасних ОС являють собою добре структуровані модульні системи, здатні до розвитку, розширення та переносу на нові платформи. Будь-якої єдиної уніфікованої архітектури ОС не існує, але відомі універсальні підходи до структурування ОС. Принципово важливими універсальними підходами до розробки архітектури ОС є:

- модульна організація;
- функціональна надлишковість;
- функціональна вибірковість;
- параметрична універсальність;
- концепція багаторівневої ієрархічної обчислювальної системи, за якою ОС є багатошаровою структурою;
- поділ модулів на дві групи за функціями: ядро-модулі, що виконують основні функції ОС, і модулі, що виконують допоміжні функції ОС;
- поділ модулів ОС на дві групи за розміщенням у пам'яті обчислювальної системи: *резидентні*, які постійно перебувають в оперативній пам'яті, і *транзитні*, що завантажуються в оперативну пам'ять тільки на період виконання своїх функцій;
- реалізація двох режимів роботи обчислювальної системи: привілейованого режиму (режиму ядра – *Kernel mode*), або режиму супервізора (*supervisor mode*) і режиму користувача (*user mode*), або режиму завдання (*task mode*);
- обмеження функцій ядра (а отже, і кількості модулів ядра) до мінімальної кількості необхідних найважливіших функцій.

Перші ОС розроблялися як монолітні системи без чітко вираженої структури (рис. 1.3).

Для побудови монолітної системи необхідно скомпілювати всі окремі процедури, а потім з'єднати їх разом в єдиний об'єктний файл за допомогою компоувальника (прикладом можуть слугувати ранні версії ядра UNIX або Novell NetWare). Кожна процедура вбачає будь-яку іншу процедуру (на відміну від структури, що містить модулі, в якій більша частина інформації є локальною для модуля, і процедури модуля можна викликати тільки через спеціально визначені точки входу).

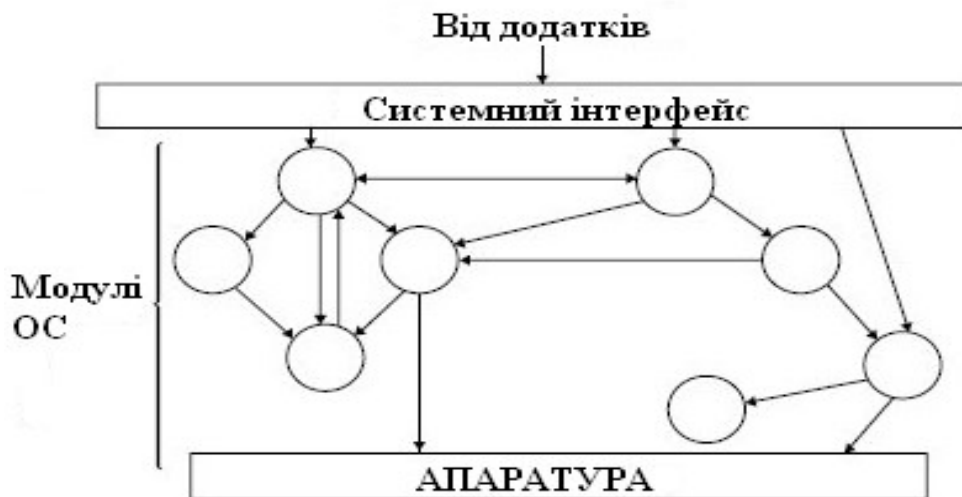


Рис. 1.3. Монолітна архітектура

Але навіть такі монолітні системи можуть бути трохи структурованими. У разі звернення до системних викликів, що підтримуються ОС, параметри розміщуються в чітко визначені місця, такі як реєстри або стек, а потім виконується спеціальна команда переривання, відома як виклик ядра чи виклик супервізора. Ця команда перемикає машину з режиму користувача в режим ядра, званий також режимом супервізора, і передає управління ОС. Потім ОС перевіряє параметри виклику, щоб визначити, який системний виклик повинен бути виконаний. Після цього ОС індексує таблицю, яка містить посилання на процедури, і викликає відповідну процедуру. Схема структурованої архітектури подана на рисунку 1.4.

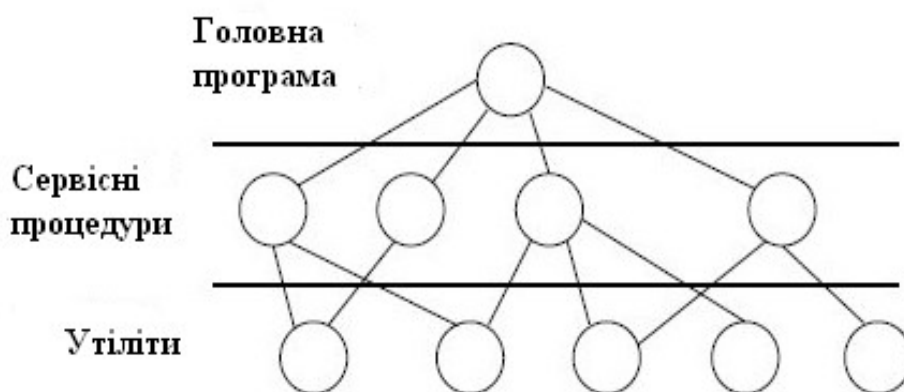


Рис. 1.4. Структурована архітектура

У цій моделі для кожного системного виклику є одна сервісна процедура. Утиліти виконують функції, які потрібні декільком сервісним процедурам.

Класичною вважається архітектура ОС, заснована на концепції ієрархічної багаторівневої машини, привілейованому ядрі і режимі користувача роботи транзитних модулів. Модулі ядра виконують базові функції ОС: управління процесами, пам'яттю, пристроями введення / виведення тощо. Ядро становить серцевину ОС, без якої вона є повністю непрацездатною і не може виконати жодну зі своїх функцій. У ядрі вирішуються внутрішньосистемні задачі організації обчислювального процесу, недоступні для додатку.

Особливий клас функцій ядра слугує для підтримки додатків, створюючи для них так зване прикладне програмне середовище. Додатки можуть звертатися до ядра із запитом – системними викликами – для виконання певних дій, наприклад відкриття та читання файлу, отримання системного часу, виведення інформації на дисплей та ін. Функції ядра, які можуть викликатися додатками, утворюють інтерфейс прикладного програмування – API (Application Programming Interface).

Для забезпечення високої швидкості роботи ОС модулі ядра (принаймні більша їх частина) є резидентними і працюють у привілейованому режимі (Kernel mode). Цей режим, по-перше, повинен забезпечити безпечну роботу самої ОС від втручання додатків і, по-друге, можливість роботи модулів ядра з повним набором машинних інструкцій, що дозволяють, власне, ядру виконувати управління ресурсами комп'ютера, зокрема перемикання процесора із задачі на задачу, управління пристроями введення/виведення, розподілом і захистом пам'яті та ін.

Решта модулів ОС виконують не настільки важливі функції, як ядро, і є транзитними. Наприклад, це можуть бути програми архівування даних, дефрагментації диска, стиснення дисків, очищення дисків тощо.

Допоміжні модулі зазвичай поділяються на групи:

- *утиліти* – програми, які виконують окремі завдання управління і супроводу обчислювальної системи;

- *системні опрацьовуючі програми* – текстові та графічні редактори (Paint, Imaging в Windows), компілятори та ін.;
- *програми надання користувачу додаткових послуг* (спеціальний варіант для користувача інтерфейсу, калькулятор, ігри, засоби мультимедіа Windows);
- *бібліотеки процедур* різного призначення, що спрощують розробки додатків, наприклад бібліотека функцій введення / виведення, бібліотека математичних функцій тощо.

Ці модулі ОС оформляються як звичайні додатки, звертаються до функцій ядра за допомогою системних викликів і виконуються в режимі користувача (user mode). У цьому режимі забороняється виконання деяких команд, які пов'язані з функціями ядра ОС (управління ресурсами, розподіл і захист пам'яті та ін.).

У концепції багаторівневої (багатошарової) ієрархічної машини структура ОС також є рядом шарів. За такої організації кожен шар обслуговує вищерозміщений шар, виконуючи для нього деякий набір функцій, які утворюють міжшаровий інтерфейс. На основі цих функцій наступний верхній за ієрархією шар будує свої функції: більш складні і більш потужні та ін. Така організація системи істотно спрощує її розробку, тому дозволяє спочатку «зверху вниз» визначити функції шарів і міжшарові інтерфейси, а за детальної реалізації, рухаючись «знизу вгору», – нарощувати потужність функцій шарів. Крім того, модулі кожного шару можна змінювати без необхідності змін в інших шарах (але не змінюючи міжшарових інтерфейсів).

Багатошарова структура ядра ОС може бути представлена, наприклад, варіантом, показаним на рисунку 1.5.

У цій схемі виділені такі шари.

Засоби апаратної підтримки ОС. Як правило, в сучасних системах завжди є засоби апаратної підтримки ОС, які прямо беруть участь в організації обчислювальних процесів. До них належать: система переривань, засоби підтримки привілейованого режиму, засоби підтримки віртуальної пам'яті, системний таймер, засоби перемикання контекстів процесів (інформація про

стан процесу в момент його припинення), засоби захисту пам'яті та ін.



Рис. 1.5. Багатошарова структура ОС

Машинно-залежні модулі ОС. Відображається специфіка апаратної платформи комп'ютера. Призначення цього шару – «екранування» верхніх шарів ОС від особливостей апаратури (наприклад, у Windows – це шар HAL (Hardware Abstraction Layer), рівень апаратних абстракцій).

Базові механізми ядра. Цей шар модулів виконує найбільш примітивні операції ядра: програмне переключення контекстів процесів, диспетчеризацію переривань, переміщення сторінок між основною пам'яттю і диском тощо. Модулі цього шару не приймають рішень про розподіл ресурсів, а тільки обробляють рішення, прийняті модулями вищерозміщених рівнів.

Менеджери ресурсів. Модулі цього шару виконують стратегічні завдання з управління ресурсами обчислювальної системи. Це менеджери (диспетчери) процесів введення / виведення,

оперативної пам'яті і файлової системи. Кожен менеджер веде облік вільних і використовуваних ресурсів і планує їх розподіл відповідно запитам додатків.

Інтерфейс системних викликів. Це верхній шар ядра ОС, який взаємодіє з додатками і системними утилітами, він утворює прикладний програмний інтерфейс ОС. Функції API, що обслуговують системні виклики, надають доступ до ресурсів системи в зручній компактній формі, без зазначення деталей їхнього фізичного розташування.

Для підвищення стійкості і надійності ОС у сучасних системах передбачений привілейований режим ядра, при якому відбувається деяке уповільнення виконання системних викликів. Системний виклик привілейованого ядра ініціює перемикання процесора з режиму користувача в привілейований, а під час повернення до додатка – зворотнє перемикання. Недоліком такого підходу є виникнення додаткової затримки в обробці системного виклику, але таке рішення стало класичним і використовується в багатьох ОС (UNIX, VAX, VMS, IBM OS/390, OS/2 та ін.).

Недоліки багат шарової класичної багаторівневої архітектури ОС: 1) значні зміни одного з рівнів можуть призвести до непередбачуваного впливу на суміжні рівні; 2) численні взаємодії між сусідніми рівнями ускладнюють забезпечення безпеки.

Тому як альтернатива класичному варіанту архітектури ОС, часто використовується *мікроядерна архітектура ОС*. За такої архітектури ОС у привілейованому режимі залишається працювати тільки досить невелика частина ОС, що називається *мікроядром*.

Мікроядро захищене від інших частин ОС і додатків. До його складу входять машинно-залежні модулі, а також модулі, що виконують базові механізми звичайного ядра. Усі інші більш високорівневі функції ядра оформляються як модулі, що працюють у режимі користувача. Так, менеджери ресурсів, які є невід'ємною частиною звичайного ядра, стають «периферійними» модулями, що працюють у режимі користувача. Отже, в архітектурі з мікроядром традиційне розташування рівнів по вертикалі замінюється горизонтальним (рис. 1.6).

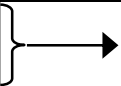
РЕЖИМ ЯДРА	Утиліти ОС, додатки	РЕЖИМ КОРИСТУВАЧА	Додатки користувачів	Утиліти	API	Сервер безпеки	Файлова система	Драйвер пристроїв
	Інтерфейс системи введення / виведення							
	Менеджер віртуальної пам'яті							
	Менеджер процесів							
	Базові механізми ядра							
	Машинно-залежні модулі							
Засоби апаратної підтримки ядра			МІКРОЯДРО (режим ядра)					
Засоби апаратної підтримки ядра			Засоби апаратної підтримки ядра					
АПАРАТУРА		АПАРАТУРА						

Рис. 1.6. Перехід до мікроядерної архітектури

Зовнішні щодо мікроядра компоненти ОС реалізуються як обслуговуючі процеси. Між собою вони взаємодіють як рівноправні партнери за допомогою обміну повідомленнями, які передаються через мікроядро. Оскільки призначенням цих компонентів ОС є обслуговування запитів додатків користувачів, утиліт і системних обробляючих програм, менеджери ресурсів, винесені в користувацький режим, називаються серверами ОС, тобто модулями, основним призначенням яких є обслуговування запитів локальних додатків та інших модулів ОС.

Механізм звернення до функцій ОС у вигляді серверів формує клієнт-серверну архітектуру (рис. 1.7).



Рис. 1.7. Клієнт-серверна архітектура

У багатьох джерелах визнані переваги мікроядерної архітектури: однотипні інтерфейси, простота розширюваності, висока гнучкість, висока надійність, підтримка розподілених систем, підтримка об'єктно-орієнтованих ОС.

У сучасних операційних системах розрізняють такі *види ядер*.

Наноядро. Досить спрощене і мінімальне ядро виконує лише одну задачу – обробку апаратних переривань, що генеруються пристроями комп'ютера. Після обробки посилає інформацію про результати обробки вищерозміщеному програмному забезпеченню. Наноядро використовується для віртуалізації апаратного забезпечення реальних комп'ютерів.

Мікроядро надає тільки елементарні функції управління процесами і мінімальний набір абстракцій для роботи з обладнанням. Велика частина роботи здійснюється за допомогою спеціальних користувацьких процесів – сервісів. У мікроядерній операційній системі можна, не перериваючи її роботу, завантажувати і вивантажувати нові драйвери, файлові системи тощо. Мікроядерними є ядра ОС Minix та GNU Hurd і ядро систем сімейства BSD. Класичним прикладом мікроядерної системи є Symbian OS. Це приклад поширеної і відпрацьованої мікроядерної (починаючи з версії Symbian OS v8.1, і наноядерної) операційної системи.

Екзоядро надає лише набір сервісів для взаємодії між додатками, а також необхідний мінімум функцій, пов'язаних із захистом: виділення і вивільнення ресурсів, контроль прав доступу та ін. Екзоядро не займається наданням абстракцій для фізичних ресурсів – ці функції виносяться в бібліотеку користувацького рівня (так звану libOS). Мікроядра ОС, що базуються на екзоядрі, забезпечують більшу ефективність за рахунок відсутності потреби у перемиканні між процесами у разі кожного звернення до обладнання.

Монолітне ядро надає широкий набір абстракцій обладнання. Усі частини ядра працюють в одному адресному просторі. Монолітне ядро вимагає перекомпіляції під час зміни складу

обладнання. Компоненти операційної системи є не самостійними модулями, а складовими частинами однієї програми. Таке ядро більш продуктивне, ніж мікроядро, оскільки працює як один великий процес. Монолітним ядром є більшість Unix-систем і Linux. Монолітність ядер ускладнює налагодження, розуміння коду ядра, додавання нових функцій і можливостей, видалення непотрібного, успадкованого від попередніх версій коду. «Розбухання» коду монолітних ядер також підвищує вимоги до обсягу оперативної пам'яті.

Модульне ядро – сучасна вдосконалена модифікація архітектури мікроядра. На відміну від «класичних» ОС з архітектурою мікроядра, модульні ядра не вимагають повної перекомпіляції ядра під час зміни складу апаратного забезпечення комп'ютера. Замість цього вони надають певний механізм підвантаження модулів, що підтримують певне апаратне забезпечення (наприклад, драйверів). Підвантаження модулів може бути як динамічним, так і статичним (у разі перезавантаження ОС після переконфігурації системи). Модульне ядро більш зручне для розробки, ніж традиційні монолітні ядра. Вони надають програмний інтерфейс (API) для з'єднання модулів з ядром, для забезпечення динамічного підвантаження і вивантаження модулів. Не всі частини ядра можуть бути зроблені модулями. Деякі частини ядра завжди зобов'язані бути присутніми в оперативній пам'яті і повинні бути жорстко «вшиті» в ядро.

Гібридне ядро – модифіковані мікроядра, що дозволяють для прискорення роботи запускати «несуттєві» частини в просторі ядра. Мають «гібридні» переваги і недоліки. Прикладом змішаного підходу може слугувати можливість запуску операційної системи з монолітним ядром під управлінням мікроядра. Мікроядро забезпечує управління віртуальною пам'яттю і роботу низькорівневих драйверів. Усі інші функції, зокрема взаємодія з прикладними програмами, здійснюються монолітним ядром. Цей підхід сформувався в результаті спроб використати переваги мікроядерної архітектури, зберігаючи за можливості добре налагоджений код монолітного ядра.

Найбільш тісно елементи мікроядерної архітектури та елементи монолітного ядра переплетені в ядрі Windows NT. Хоча Windows NT часто називають мікроядерною операційною системою, це не зовсім так. Мікроядро NT надто велике (більше 1 Мбайт), щоб носити приставку «мікро». Компоненти ядра Windows NT розташовуються в пам'яті і взаємодіють один з одним шляхом передачі повідомлень, як і належить у мікроядерних операційних системах. Водночас усі компоненти ядра працюють в одному адресному просторі й активно використовують загальні структури даних, що властиво операційним системам з монолітним ядром.

1.5.1. Архітектура ОС Windows

На рисунку 1.8 представлена загальна структура Windows ¹.

Усі версії Windows основані на NT версії і мають в основному подібну структуру на цьому рівні деталізації. У Windows відділено програмне забезпечення, що орієнтоване на користувача ОС від програмного забезпечення ядра ОС. Останнє, яке включає виконавчу систему (Windows Executive), ядро, драйвери пристроїв, рівень абстрагування від устаткування (HAL, hardware abstraction layer), запускається в режимі ядра. Програмне забезпечення режиму ядра ОС має доступ до системних даних і до устаткування комп'ютера. Усе інше програмне забезпечення, яке запускається в режимі користувача, має обмежений доступ до системних даних.

Організація ОС Windows має високорозвинену модульну архітектуру, в якій кожна системна функція є керованою тільки одним компонентом ОС, а вся інша частина ОС і всі додатки отримують доступ до цієї функції лише через дублюючий компонент, який використовує стандартні інтерфейси. Доступ до

¹ Stallings, William. Operating systems: internals and design principles / William Stallings. – 7th ed. Prentice Hall, New Jersey, 2012, p. 769. ISBN-13:978-0-13-230998-1

ключових системних даних може бути отримано лише через відповідні функції.

У принципі будь-який модуль може бути вилючено, модифіковано чи переміщено без перезапису всієї системи або її стандартних програмних інтерфейсів додатків (API, Application Programming Interface).

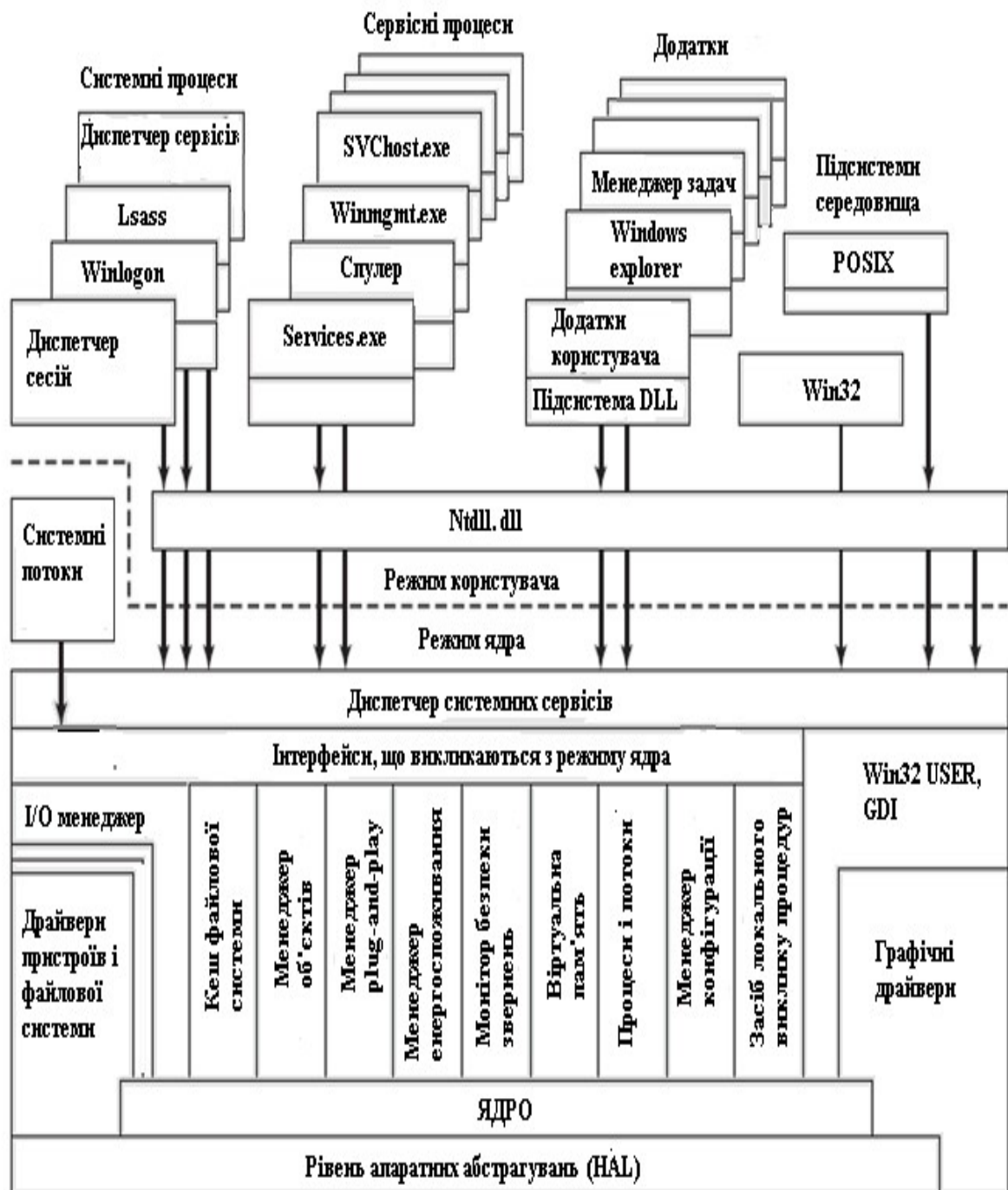


Рис. 1.8. Архітектура Windows

Як бачимо із схеми на рисунку 1.8, до компонентів режиму ядра належать:

- *виконавча система*, що складається із шару сервісів ОС, таких як управління пам'яттю, управління процесами і потоками, безпеки, системи введення / виведення;
- *рівень апаратних абстракцій*, який відокремлює ОС від особливостей апаратної платформи, і тому системна шина, контролер прямого доступу до пам'яті, контролер переривань, системні таймери і пам'ять з точки зору ядра виглядають однаково;
- *ядро Windows* відповідає за розподіл ресурсів між процесами, їх перемикання і синхронізацію; основним завданням ядра є якомога більш ефективно 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);
- *драйвери пристроїв* можуть бути драйвери не тільки апаратних пристроїв, але і застосування, якому потрібні засоби, доступні в режимі ядра, і які завжди варто оформляти як драйвер;
- *віконна і графічна підсистеми* відповідають за інтерфейс користувача – роботу з вікнами, елементами керування і графічним виведенням.

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

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

- *Менеджер об'єктів* – відповідає за розподіл ресурсів у системі, підтримуючи їх універсальне подання через об'єкти.
- *Засіб локального виклику процедур (ALPC, advanced local procedure call)* – забезпечує механізм зв'язку між процесами і підсистемами на одному комп'ютері.
- *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.
- *Менеджер plug-and-play* – менеджер підключення, відключення, налаштування пристроїв, які підтримують технологію Plug-and-Play.

1.5.2. Особливості сучасних архітектур ОС UNIX і Linux

Архітектура ОС UNIX. Сучасна ОС UNIX має порівняно просту архітектуру. В ядрі містяться найбільш вживані функції, що забезпечують роботу ОС і користувача. Ядро зі свого боку взаємодіє з прикладними програмами за допомогою системних викликів. Сучасна структура ядра UNIX може бути представлена, як зображено на рисунку 1.9. В архітектурі сучасних UNIX-систем 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.

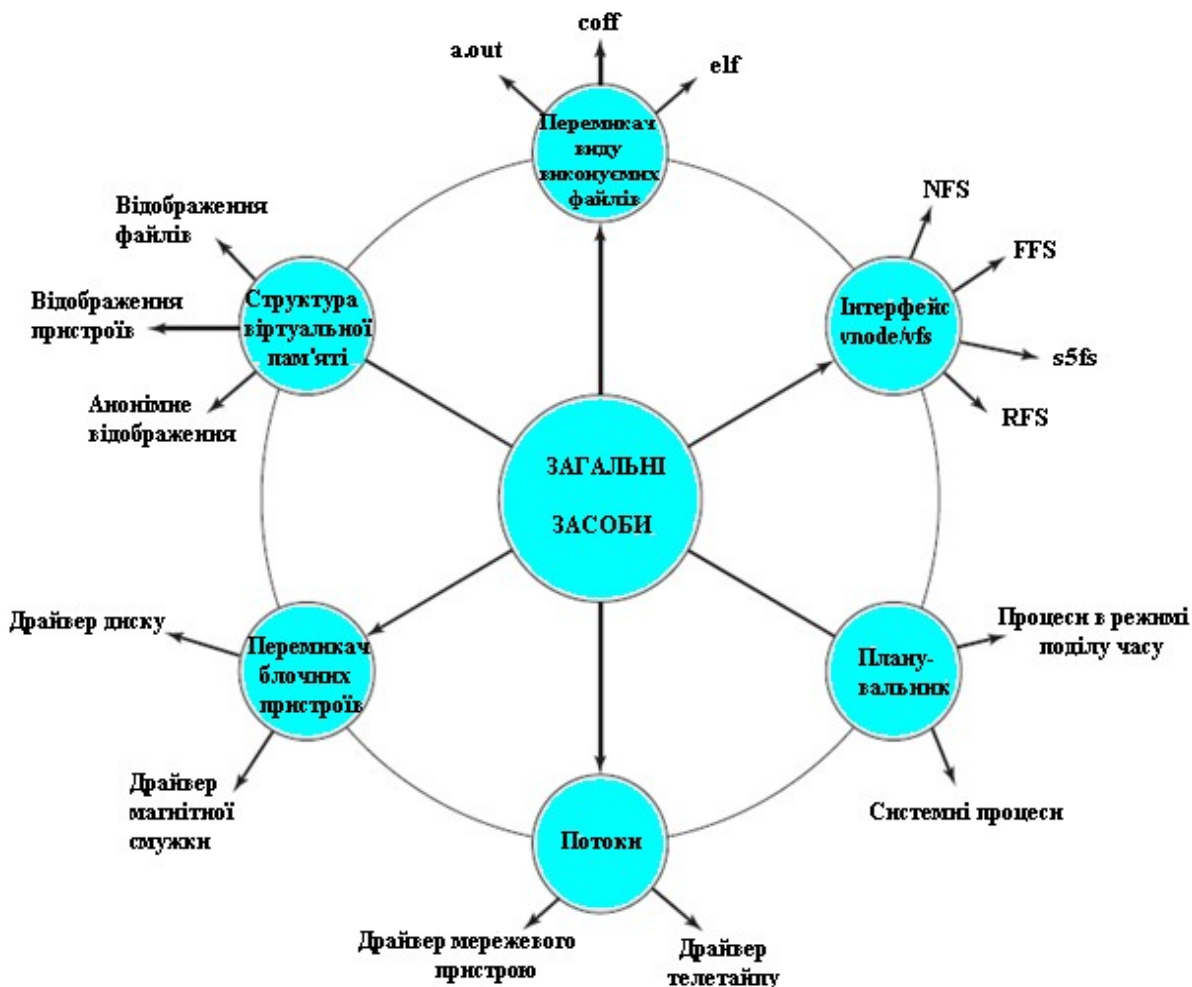


Рис. 1.9. Ядро сучасної системи UNIX

Архітектура Linux. Зауважимо, що архітектура Linux організована за технологією монолітного ядра, тобто структура даних ядра і код перебувають в єдиному адресному просторі.

На рисунку 1.10 показані основні компоненти ядра Linux, які впроваджені в IA-64 архітектуру (тобто Intel Itanium). Прямокутники з хвилястими лініями, що у верхній частині схеми, – це окремі процеси з потоками в них.

Саме ядро складається з набору взаємодіючих компонентів, між якими показано основні взаємодії. Нижче розміщена апаратна частина, що представлена як набір компонентів. Стрілками показано, які компоненти ядра використовують або керують

Розділ 1. Архітектура, призначення і функції операційних систем

компонентами апаратури. Усі компоненти ядра виконуються процесором, але це на схемі не показано.

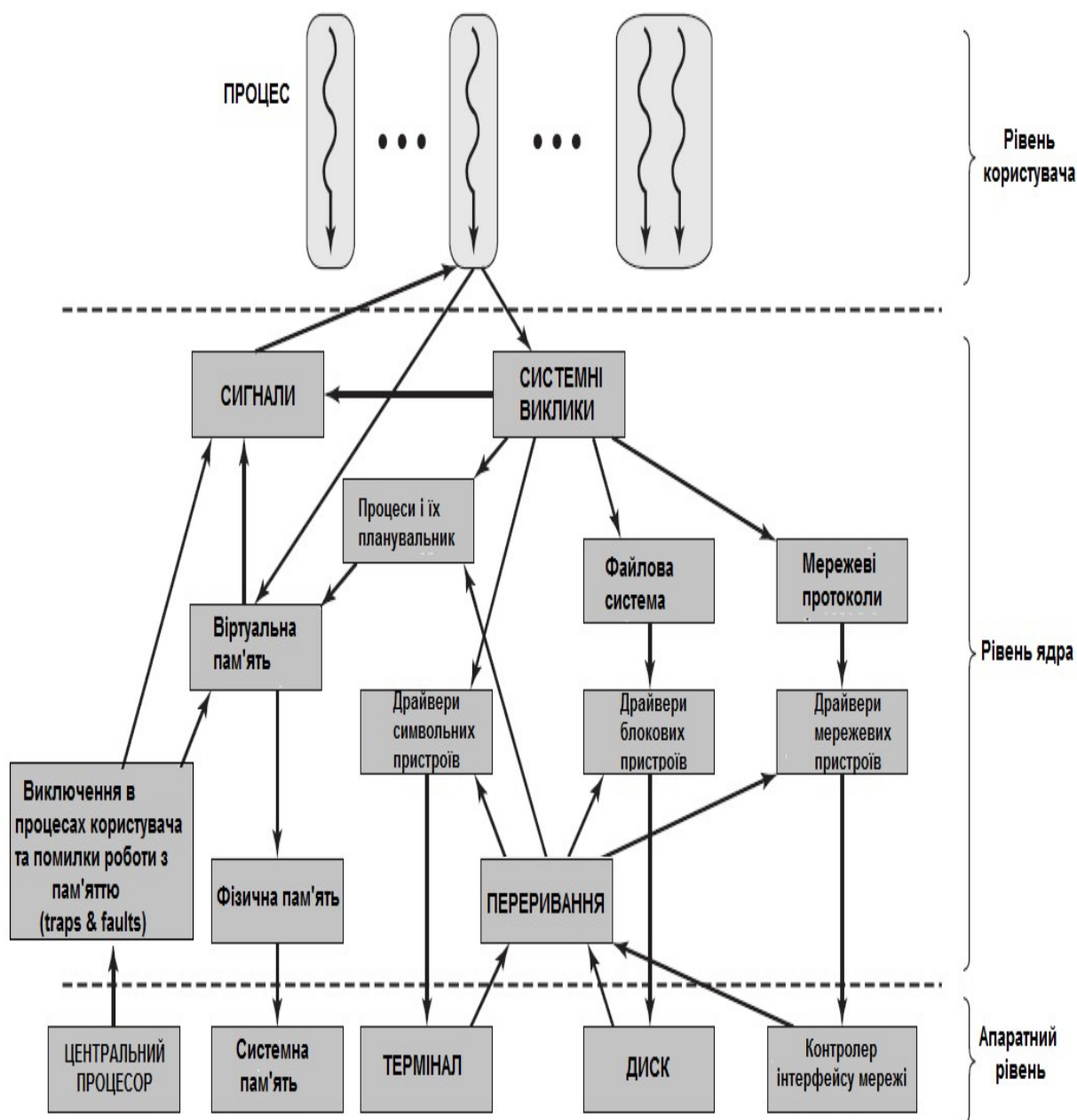


Рис. 1.10. Основні компоненти ядра Linux

Основні функціональні компоненти ядра:

- *менеджер пам'яті* – організація адресного простору і віртуальної пам'яті;
- *планувальник процесів* – реалізує підтримку багатозадачності в операційній системі: робота з таймером, створення процесів, завершення процесів та ін.;

- *міжпроцесова взаємодія* – механізми, що забезпечують обмін даними між процесами;
- *віртуальна файлова система* – інтерфейс взаємодії з різними файловими системами та пристроями введення / виведення. Забезпечує доступ до драйверів пристроїв, що забезпечують безпосередню роботу з периферійними пристроями;
- *мережний інтерфейс* – забезпечує доступ до реалізації мережних протоколів і драйверів мережних пристроїв.

У Linux підтримується так звана концепція *модулів ядра*, тобто комплекс незалежних блоків. Ці блоки володіють, поперше, *динамічним з'єднанням*, при якому на вимогу завантажуються у пам'ять і вивантажуються з неї окремі секції коду. Такі секції (модулі) виконуються у привілейованому режимі. І, по-друге, *стековою організацією*, при якій модулі організовані у вигляді певної ієрархічної структури. Окремі модулі виконують роль бібліотек у процесі звернення до них модулів більш високих рівнів у межах цієї структури, і вони самі можуть звертатися до модулів на більш низьких рівнях.

Для підтримки модулів у Linux використовують певні засоби: засоби, що завантажують модулі у пам'ять із здійсненням обміну даними між модулями та іншою частиною ядра; реєстрація драйверів для попередження іншої частини ядра про те, що новий драйвер став доступним; резервування апаратних ресурсів для захисту драйверів пристроїв від випадкового використання їх іншими драйверами.

1.6. Операційна система Android

Операційна система Android – це система на основі Linux, спочатку розроблена для мобільних телефонів. Це найпопулярніша мобільна ОС з великим відривом: телефони Android перевершують iPhone від Apple приблизно в 4 рази [Morra, J. Google Rolls Out New Version of Android Operating System. Electronic Design, August 24, 2016]. Але це тільки один з елементів зростаючого домінування Android. Усе частіше ця ОС стоїть практично за

будь-яким пристроєм з комп'ютерним чіпом, крім серверів і ПК. Android – широко використовувана ОС для інтернет-речей.

Первісна розробка ОС Android була здійснена компанією Android, Inc., яка була куплена Google 2005 року. Перша комерційна версія Android 1.0 була випущена 2008 року. На момент написання цього посібника останньою версією є Android 7.0 (Nougat). Android має активне співтовариство розробників та ентузіастів, які використовують вихідний код Android Open Source Project (AOSP) для розробки і поширення власних модифікованих версій операційної системи. Відкритий характер Android був ключем до його успіху.

1.6.1. Архітектура програмного забезпечення Android

Android визначається як програмний стек, який включає модифіковану версію ядра Linux, проміжне програмне забезпечення і ключові програми. На рисунку 1.11 подано архітектуру програмного забезпечення Android у деяких деталях. Отже, Android варто розглядати як повний програмний стек, а не просто як ОС.

Додатки. Усі додатки, з якими користувач взаємодіє безпосередньо, є частиною рівня додатків. Сюди входить основний набір додатків загального призначення, таких як поштовий клієнт, програма SMS, календар, карти, браузер, контакти та інші додатки, які зазвичай використовуються на будь-якому мобільному пристрої. Додатки зазвичай реалізуються на Java. Основна мета архітектури Android з відкритим вихідним кодом полягає в тому, щоб полегшити розробникам реалізацію нових додатків для конкретних пристроїв і вимог кінцевого користувача. Використання Java дозволяє розробникам позбутися специфічних для апаратного забезпечення міркувань і особливостей, а також використовувати більш широкі можливості мови Java, такі як зумовлені класи. Приклади типів базових додатків на платформі Android: Google Keep – нотатки та списки; Facebook, WhatsApp Messenger, Instagram; Google Новини; Microsoft Word: правка документів та загальний доступ; Google Документи та ін.

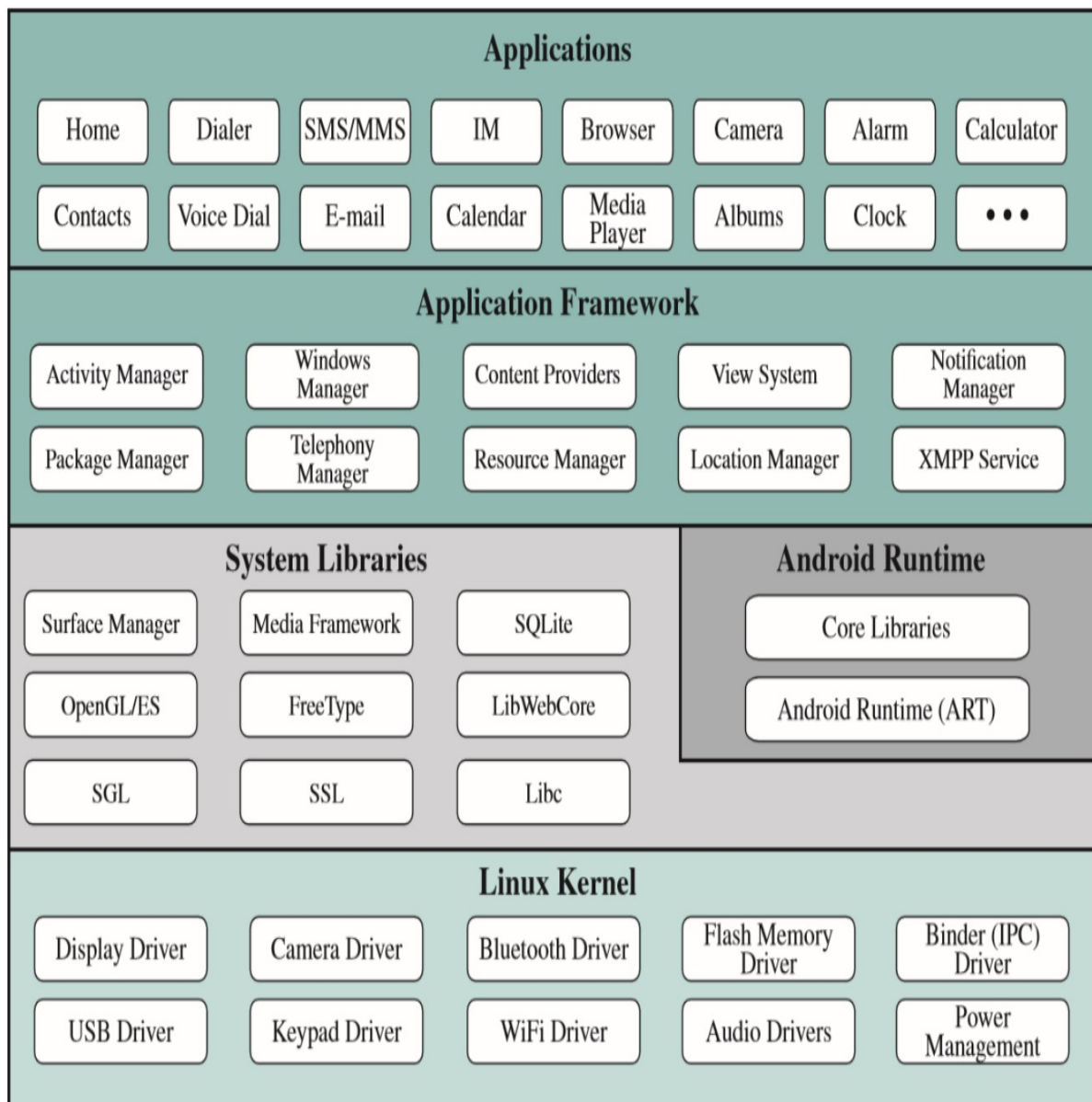


Рис. 1.11. Архітектура програмного забезпечення Android

Пояснення до рисунка 1.11:

Applications – рівень додатків;

Applications Framework – рівень спеціалізованих бібліотек для додатків;

System Libraries – системні бібліотеки;

Android Runtime – середовище виконання Android-додатків з їх компіляцією під час установки;

Linux Kernel – ядро ОС Linux.

1.7. Архітектура iOS

Розробником iOS є Apple Inc. Ця ОС належить до UNIX подібних операційних систем на основі компонентів Darwin¹. В основі цієї ОС лежать мови програмування C, C++, Objective-C, Swift. Загальний вихідний код є закритим, але окремі компоненти можуть бути відкритими.

Перший випуск операційної системи iOS відбувся 27 червня 2007 року. Фінальні версії оновлюються один раз на рік. Ця операційна система, крім англійської мови, підтримує ще 39 мов.

На рисунку 1.12 подано спрощену архітектуру iOS (набір інтерфейсів).

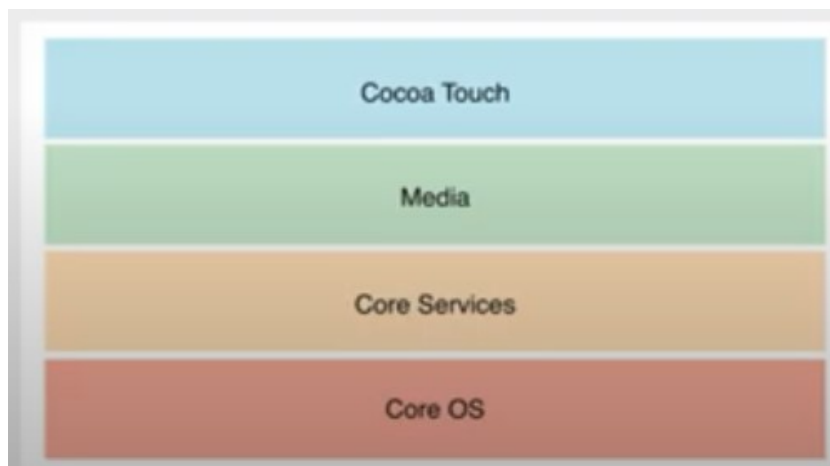


Рис. 1.12. Спрощена архітектура iOS (набір інтерфейсів)

OS-технології можна розглядати як набір шарів, де Cocoa Touch перебуває на найвищому рівні, а Core OS та ядро Mac OS X – на нижчих.

Це дозволяє реалізовувати багато складних завдань, скорочуючи обсяг роботи, яку довелося б виконувати розробникам,

¹ Darwin – це відкрита POSIX-сумісна ОС, створена Appl Inc. 2000 року. Вона об'єднує коди, написані Apple, NeXTSTEP, FreeBSD, та інших вільних проєктів. Система Darwin являє собою набір основних компонентів, що використовуються в macOS та iOS. Є сумісною з третьою версією специфікації єдиної UNIX (SUSv3) та POSIX-додатками й утилітами.

починаючи з нижнього рівня. Проте деякі низькі шари абстрагування можуть бути доступні розробникам за необхідності.

Розташування шарів абстрагування від найвищого до нижчого представлено на рисунку 1.13.

Cocoa – рідне об'єктно-орієнтоване середовище розробки програм для операційної системи Mac OS X виробництва компанії Apple. Це один із п'яти основних API, доступних у Mac OS X: Cocoa, Carbon, Toolbox (для роботи старих програм Mac OS 9), POSIX і Java. Такі мови, як Perl, Python і Ruby, не вважаються основними, тому що на них досі ще пишеться не так багато серйозних програм для Mac OS X.

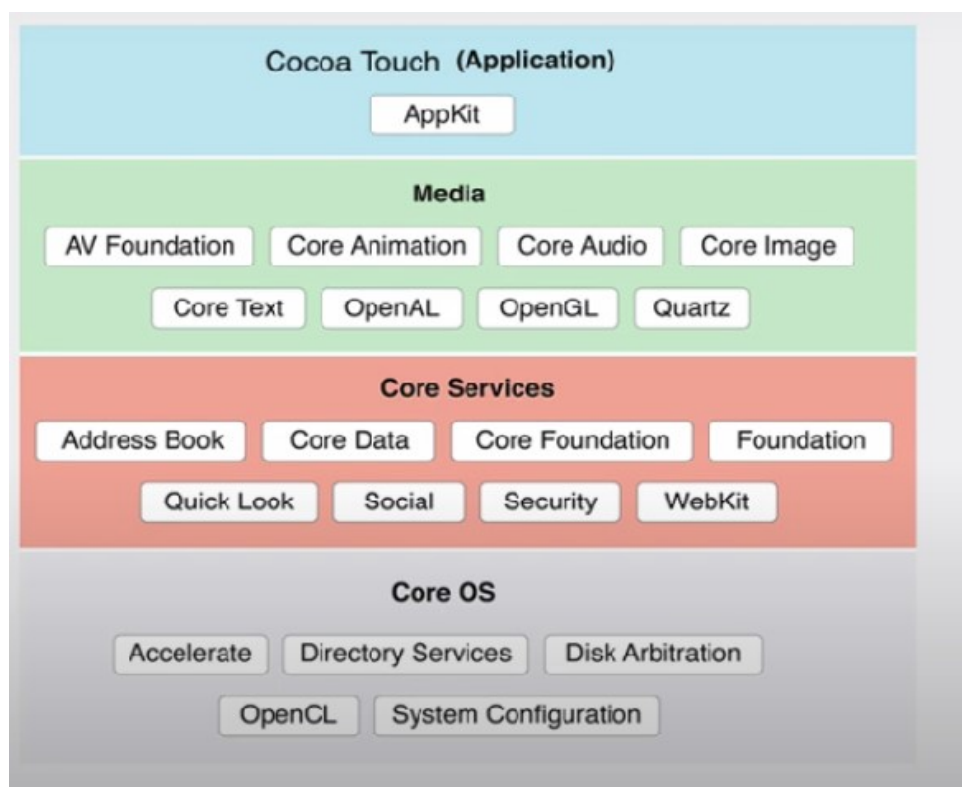


Рис. 1.13. Розташування шарів абстрагування від найвищого до нижчого

Програми, що використовують Cocoa, зазвичай розробляються за допомогою середовища розробки Apple Xcode (у минулому називався Project Builder) та Interface Builder з використанням мови Objective-C. Однак середовище Cocoa також доступне і під час розробки іншими мовами, такими як Ruby,

Python і Perl за допомогою сполучних бібліотек (MacRuby, PyObjC та CamelBones, відповідно). Також можна створювати Cocoa-програми на Objective-C у звичайному текстовому редакторі та вручну компілювати їх за допомогою GCC або make-сценаріїв для GNUstep.

З точки зору кінцевого користувача, Cocoa-програми – це програми, написані з використанням програмного середовища Cocoa. Такі програми зазвичай мають характерний зовнішній вигляд, оскільки це середовище значно спрощує підтримку принципів «людського інтерфейсу» Apple (Apple Human Interface Guidelines).

У Cocoa Touch закладено основні технології та можливості: Core Animation, мультизадачність, розпізнавачі мультитач-жестів.

Зазначимо, що Cocoa Touch надає основні фреймворки для розробки додатків на пристроях під управлінням iOS.

Наприклад:

- ✓ Foundation Kit Framework – основна бібліотека, що містить класи з префіксом NS;
- ✓ UIKit Framework (основується на Application Kit) – бібліотека, що містить специфічні для iOS GUI-класи;
- ✓ Game Kit Framework – бібліотека для взаємодії із сервісом Game Center;
- ✓ iAd Framework – бібліотека для реалізації сервісів контекстної реклами iAd усередині додатків;
- ✓ Map Kit Framework – бібліотека, що здійснює взаємодію з картами та навігаційними можливостями iOS-пристроїв.

1.8. Основні етапи розвитку операційних систем

До середини 1950-х років перші ЕОМ ще не були забезпечені операційними системами.

На початку 1960-х вони лише комплектувались набором інструментів для розробки, планування та виконання завдань і до кінця 1960-х було розроблено ряд операційних систем. Найбільш розвинені ОС того часу, такі як «OS/360» (компанія «IBM»), «SCOPE» (компанія «CDC») та завершений уже в 1970-х роках

«MULTICS» (MTI та компанія «Bell Labs»), передбачали можливість використання багатопроцесорних систем.

Спонтанний характер розробки ОС призвів до наростання кризових явищ, пов'язаних насамперед зі складністю та великими розмірами розроблюваних систем. ОС погано масштабувались (більш прості не використовували всіх можливостей потужних обчислювальних машин; складніші неоптимально виконувались або взагалі не виконувались на менш потужних системах) і були повністю несумісними між собою.

1969 року співробітники MTI Кен Томпсон, Деніс Рітчі та Браян Керніган з колегами розробили та реалізували ОС «Юнікс» («Unix»; первинно «UNICS», на противагу «MULTICS»). Нова ОС увібрала в себе багато рис попередниць, але на противагу їм мала ряд переваг:

- проста метафорика (два ключових поняття – процес та файл);
- компонентна архітектура (принцип «одна програма – одна функція», або інакше «кожна програма має робити лише одну роботу, але робити її добре» плюс потужні засоби об'єднання цих програм для розв'язання конкретних задач);
- мінімізація ядра та кількості системних викликів;
- незалежність від апаратної архітектури і реалізація на машинно-незалежній мові програмування (для цього була розроблена мова програмування «C»);
- уніфікація файлів (будь-що у системі є файлом, до якого можна отримати доступ за спільними для всіх правилами).

Завдяки зручності насамперед як інструментальне середовище Unix досить тепло зустріли в університетах, а потім і в галузі. У цілому і незабаром вона стала прототипом єдиної ОС, яку можна було використовувати у найрізноманітніших обчислювальних системах і більше того швидко та з мінімумом зусиль перенести на іншу апаратну архітектуру.

Наприкінці 1970-х років співробітники Каліфорнійського університету в Берклі внесли ряд суттєвих удосконалень у джерельні коди Unix, включно з реалізацією стеку мережних

протоколів TCP/IP. Їх розробка стала відомою під іменем BSD (англ. *Berkeley Software Distribution*).

Через конфлікт з «Bell Labs» Річард Столмен поставив задачу реалізувати повністю незалежну від авторських прав ОС на основі Unix, заснувавши проєкт «GNU» (англ. *рекурсивне скорочення «GNU's Not Unix»* – «ГНЮ Не Юнікс»).

Незабаром Unix стала стандартом де-факто, а потім і юридичним – ISO/IEC 9945. ОС, що дотримувались цього стандарту чи опираються на нього, називають «відкритими» або «стандартними». До них належать системи, що базуються на останній версії Unix, випущеної «Bell Labs» («System V»), на розробках Університету Берклі («FreeBSD», «OpenBSD», «NetBSD»), а також ОС «Linux», розроблена спільнотою на чолі з Лінусом Торвальдсом та в межах проєкту «GNU» (основні системні інструменти).

Нижче наведені *основні етапи в розвитку операційних систем*.

1. Прості пакетні ОС. У процесі роботи з такими ЕОМ завдання на машинних носіях збирали в пакети і поміщали в пристрій вводу даних. Після цього ОС, або монітор, зчитувала з пристрою введення / виведення ці завдання по одному і передавала управління наступному зчитаному завданню. Після завершення завдання або у разі виникнення помилки управління поверталось монітору, який починав зчитувати наступне завдання. Приклади пакетних ОС: FMS (Fortran Monitor System), IBSYS (ОС, створена фірмою IBM для ЕОМ IBM 7094).

2. Багатозадачні пакетні ОС. Для підвищення ефективності використання процесора необхідно було реалізувати принцип багатозадачності (наприклад, доки одне завдання чекає завершення виконання операції введення / виведення, інше може використовувати процесор). Це можливо, якщо в ОП є достатньо місця і для самої ОС, і для двох (або більше) програм користувача. Для багатозадачних ОС обов'язковим є:

1) наявність контролерів зовнішніх пристроїв, що працюють незалежно від процесора і дозволяють виконувати операції введення / виведення одночасно з командами процесора;

2) наявність системи переривань, за допомогою яких такі контролери взаємодіють з процесором.

Під час роботи багатозадачних ОС виникає потреба в управлінні пам'яттю, в плануванні щодо виконання задач, у диспетчеризації процесів, у захисті завдань від впливу один одного. Приклад багатозадачної пакетної ОС: OS/360 фірми IBM.

3. ОС з розподілом часу. У цьому випадку декілька користувачів одночасно отримують доступ до системи за допомогою терміналів (термінал – робоче місце користувача: дисплей і клавіатура), а ОС забезпечує чергове виконання програм кожного користувача через малі проміжки часу. Отже, кожному користувачеві надається частина процесорного часу ЕОМ, не враховуючи витрат часу на роботу ОС. Приклади ОС з розподілом часу: CTSS (Compatible Time-Sharing System), MULTICS.

Контрольні питання

1. Що таке операційна система? Які основні функції ОС?
2. На якому рівні ієрархічної структури програмно-апаратних засобів комп'ютера реалізується виконання машинних команд?
3. Назвіть сервіси, що надає ОС.
4. Як можна класифікувати ОС?
5. Як класифікуються ОС за призначенням? Відповідь деталізуйте.
6. Які основні складові ОС?
7. Що являє собою ядро ОС?
8. Що визначає структура ОС? Назвіть і опишіть основні типи структур ОС.
9. В яких режимах працюють сучасні ОС?
10. Що таке операційне і програмне середовище ОС?
11. Що таке *API*?
12. Що розуміють під архітектурою ОС? Назвіть універсальні підходи до розробки архітектури ОС.

13. Який основний недолік монолітної архітектури ОС?
14. На якій концепції заснована класична архітектура ОС?
15. Що являє собою багатошарова архітектура ОС?
16. Які особливості мікроядерної архітектури?
17. Назвіть компоненти режиму ядра ОС Windows?
18. Що називається виконавчою системою Windows (Windows Executive)?
19. Яку технологію ядра реалізує архітектура Linux?
20. Назвіть і опишіть основні етапи розвитку операційних систем.
21. Назвіть основні елементи архітектури програмного забезпечення Android.

РОЗДІЛ 2

ПРОЦЕСИ. ПОТОКИ. СИГНАЛИ¹

2.1. Основні відомості про процеси. Управління процесами

Для розуміння структури операційних систем застосовують концепцію процесів. Цей термін уперше був застосований у 60-х роках минулого століття розробниками операційної системи Multics. Існує декілька визначень терміна «процес», наприклад:

- програма, що виконується;
- екземпляр програми, що виконується на комп'ютері;
- об'єкт, який можливо ідентифікувати і виконувати на процесорі;
- одиниця активності, яка характеризується єдиним ланцюжком послідовних дій, поточним станом та з'єднаним з ним набором системних ресурсів.

2.1.1. Властивості та класифікація процесів

Більшість сучасних операційних систем є мультипрограмними².

Переваги мультипрограмного режиму:

- збільшення пропускної спроможності обчислювальної системи завдяки більш оптимальному завантаженню пристроїв;
- можливість поєднувати тривалі операції (наприклад, виведення на принтер) з іншою корисною роботою;
- нові засоби декомпозиції під час розробки складних систем, що обробляють різні асинхронні події.

¹ Під час підготовки цього розділу були використані літературні джерела [1–4], [15–18], [20–22] та інші із списку літератури.

² Мультипрограмування – це такий спосіб організації обчислень, коли на однопроцесорній обчислювальній системі імітується одночасне виконання декількох програм.

Оскільки мультипрограманість передбачає виконання декількох програм одночасно, то виникає необхідність керувати ними, розподіляючи між ними ресурси під час виконання, і, як наслідок, виникає таке поняття, як процес.

Процес – це окрема програма в момент її виконання, а також виділені їй ресурси комп'ютера.

В обчислювальних системах завжди виникають причини для створення нових процесів.

1. **Ініціалізація системи.** У разі запуску ОС відразу створюється кілька процесів, що забезпечують взаємодію з користувачем. Інші процеси є фоновими. Їх ще називають демонами (Disk And Execution MONitor).

2. **Створення процесу іншим процесом, що працює.** Створений процес є дочірнім, а процес, що його створив, – батьківським.

3. **Запит користувача на створення процесу.** Процес завжди створюється однаково за допомогою особливого системного виклику для створення нового процесу.

В обчислювальних системах завжди є ситуації, при яких процес може бути завершеним:

- 1) звичайне або успішне завершення процесу;
- 2) перевищення ліміту часу, що відведений для програми;
- 3) недостатній обсяг пам'яті;
- 4) зайве очікування (процес очікує настання певної події довше, ніж задано в параметрах системи);
- 5) вихід із процесу під час виникнення помилки, яку неможливо усунути;
- 6) поступила помилкова команда або з недосяжними привілеями;
- 7) знищення процесу іншим процесом.

У разі виконання програм процесором обчислювальної системи визначають такі стани процесора:

Passive – процес завантаження в пам'ять, але його виконання ще неініційоване;

Ready – процес готовий до виконання (очікує ресурс центрального процесора);

Run – процес виконується на центральному процесорі;

Wait – процес очікує звільнення необхідного ресурсу.

Залежної від типу очікуваного ресурсу розрізняють системи:

- з обмеженням щодо введення / виведення (в таких системах найбільший час очікування витрачається процесом під час звернення до підсистем введення / виведення);
- з обмеженням зі швидкодії (найбільший час витрачається на доступ до процесора).

Будь-який процес може перебувати в одному із станів один раз у певний момент часу (рис. 2.1).

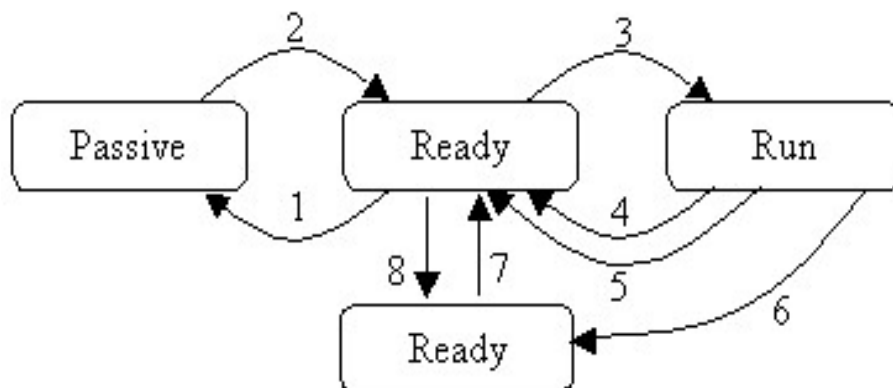


Рис. 2.1. Стани процесів

Визначають такі умови зміни станів процесу:

- 2 – активізація процесу програмою – диспетчером або ОС;
- 3 – диспетчеризація¹ процесу;
- 4 – виникнення переривань;
- 5 – запит процесом певної системної функції у складі ОС (можливе створення певного процесу);
- 6 – запит процесом певної системної функції (введення / виведення);
- 7 – завершення виконання системної функції;
- 8 – зупинка виконання процесу;
- 1 – завершення виконання процесу.

¹ Поняття диспетчеризації буде розглянуто в підрозділі «Планування і диспетчеризація процесів».

2.1.2. Моделі процесу

Модель з двома станами. Основним завданням ОС є управління виконанням процесів (визначення схеми чергування процесів та виділення їм ресурсів).

У найпростішій моделі управління процесами в будь-який момент часу процес або виконується, або не виконується. На рисунку 2.2(а) представлена модель процесу з двома станами, а на рисунку 2.2(б) – діаграма використання черги.

Створивши новий процес, ОС вводить його в систему в стані, що не виконується. Після цього створений процес чекає, коли він зможе бути запущений. Час від часу процеси, що виконуються, будуть перериватися, а диспетчер вибиратиме інший процес для виконання.

Процеси, що не виконуються, організовані в чергу (*нул*), де вони чекають свого виконання. Елементи такої черги – покажчики на процеси. Процес, робота якого була перервана, переходить у чергу процесів, що очікують виконання.

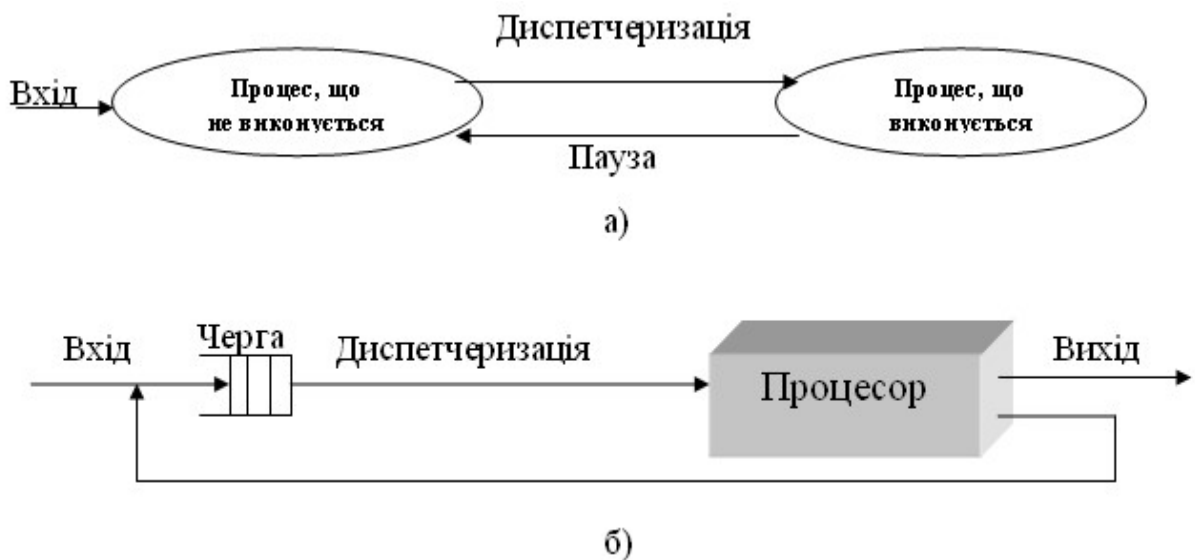


Рис. 2.2. Модель процесу з двома станами:
а) діаграма станів; б) використання черги

Якщо процес завершено, то він виводиться із системи. Далі для виконання диспетчер вибирає з черги наступний процес.

Модель з п'ятьма станами. У моделі з двома станами можливі дві некоректні ситуації, коли процес не виконується:

1) ОС вирішила на деякий час надати процесор іншому процесу;

2) процес блокований, оскільки з точки зору своєї внутрішньої логіки він не може виконуватися далі (наприклад, очікує вхідні дані).

На рисунок 2.3(а) представлені три стани процесу: 1) як такий, що виконується (використовує процесор); 2) готовий до виконання (тимчасово призупинений, щоб дозволити виконуватися іншому процесу); 3) блокований (процес не може бути запущений, доки не відбудеться деяка зовнішня подія). Тут також добавлено ще два стани: новий (тільки що створений процес, який ще не поміщений операційною системою в чергу процесів, що виконуються, і ще не завантажений в ОП) та завершений (процес, уже вилучений з пулу виконаних процесів).

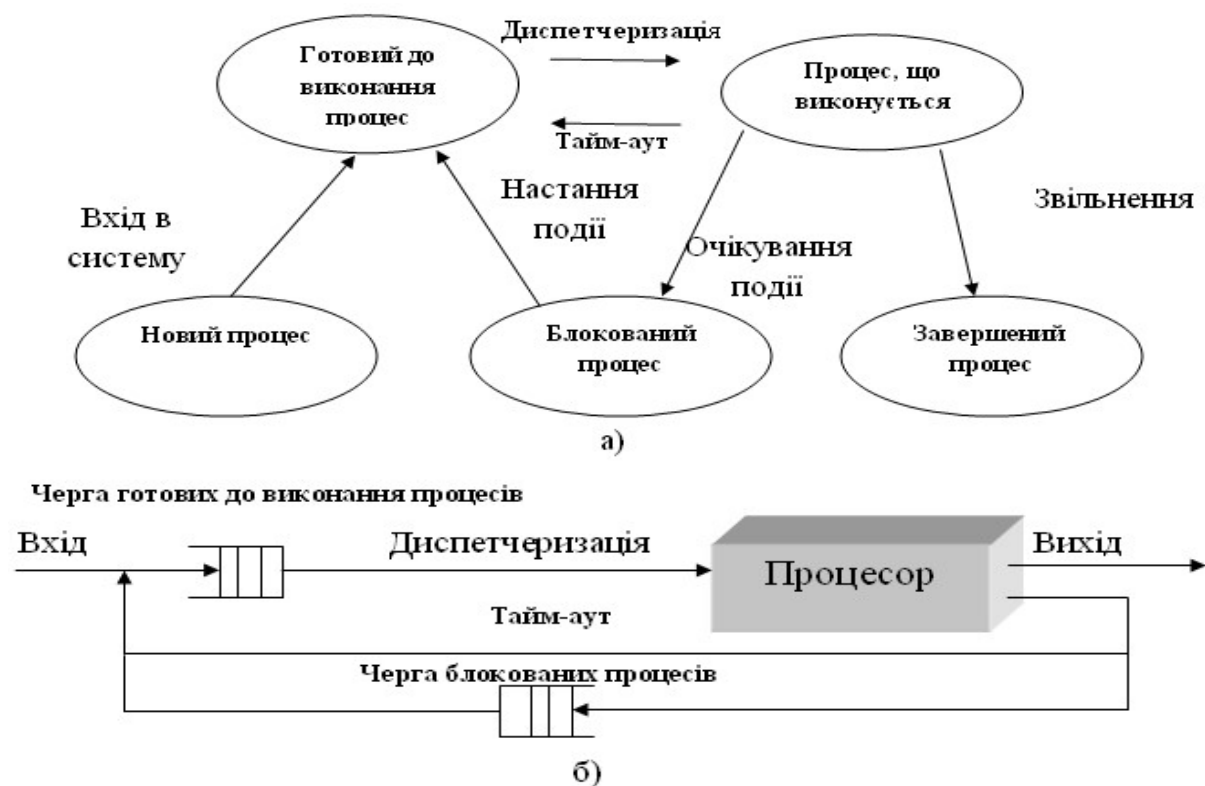


Рис. 2.3. Модель з п'ятьма станами:

а) діаграма станів; б) використання черги

З огляду на діаграми станів можливі такі переходи між станами процесу:

- **нульовий стан → новий** – причини переходу між цими станами збігаються з причинами створення процесів;
- **новий → готовий до виконання** – цей перехід відбувається, коли ОС готова до обробки додаткового процесу, оскільки у більшості ОС є обмеження на кількість процесів або на обсяг виділеної для процесів віртуальної пам'яті;
- **готовий → виконується** – цей перехід відбувається в момент вибору нового процесу для запуску;
- **виконується → завершений** – причини переходу між цими станами збігаються з причинами завершення процесів;
- **виконується → готовий до виконання** – цей перехід відбувається найчастіше після закінчення кванта часу, виділеного процесу. У деяких ОС може відбутися витіснення процесу і до закінчення цього кванта часу (наприклад, у разі появи процесу з більш високим пріоритетом). Витіснення (preemption) – це ситуація, коли процес позбавляється деякого ресурсу до того, як він завершить роботу з ним;
- **виконується → блокований** – цей перехід виконується, якщо для продовження виконання процесу потрібно дочекатися настання деякої події (звільнення зайнятого ресурсу, закінчення операції введення / виведення, повідомлення від іншого процесу тощо);
- **блокований → готовий до виконання** – цей перехід відбувається, коли настає очікувана процесом подія.

На рисунку 2.3(б) показаний один із способів реалізації по чергового виконання процесів. Маємо дві черги: 1 – для готових до виконання і 2 – для заблокованих процесів.

Кожен процес, який надходить у систему для обробки, поміщується в чергу готових до виконання процесів, і ОС вибирає з неї для виконання черговий процес. Якщо схема пріоритетів відсутня, то така черга має тип FIFO («First Input – First Output» – «першим увійшов – першим вийшов» – «класичний» варіант черги, послідовність обробки елементів якої на виході та сама, що і

послідовність поміщення цих елементів у чергу). Коли виконання процесу переривається, він, залежно від обставин, може або завершитися, або потрапити в одну з двох згаданих вище черг. Якщо ж станеться якась подія, то всі процеси, що очікують його, переміщуються з черги блокованих у чергу готових процесів.

Недоліком такої схеми є те, що після будь-якої події ОС повинна переглядати всю чергу блокованих процесів, відшуковуючи серед них ті, які очікують саме цієї події. Тому ефективніше організувати декілька черг – свою для кожної події. Тоді у разі настання будь-якої події всі процеси з відповідної черги можуть бути відразу переведені в чергу готових до виконання процесів.

Якщо ж диспетчеризація здійснюється з використанням пріоритетів¹, то можна організувати декілька черг готових процесів по одній черзі на кожен рівень пріоритету. У цьому випадку потрібно також передбачити механізм контролю виконання процесів з відповідних черг відповідно до їх пріоритету.

2.1.3. Опис процесів

Для керування процесами ОС створює і підтримує таблиці процесів, які містять показники на образ кожного процесу.

Образ процесу містить програму (код), дані, стек та атрибути процесу. Структура даних разом з атрибутами процесу часто називається його **керуючим блоком** (рис. 2.4).

Атрибути процесу включають у себе таке:

- *ідентифікатори* (ідентифікатор цього процесу, ідентифікатор батьківського процесу, ідентифікатор користувача);
- *інформацію про стан процесу* (вміст реєстрів загального призначення, керуючих реєстрів і реєстрів стану, показники на стек);
- *керуючу інформацію про процес* (стан процесу – виконується, готовий до виконання, блокований та ін.; пріоритет; інформацію, пов'язану з плануванням; інформацію про події; інформацію про зв'язки з іншими процесами (в чергах, родинні

¹ Поняття пріоритету процесу буде викладене в підрозділі 2.1.5.

зв'язки); відомості про обмін інформацією між процесами (різні прапори, сигнали, повідомлення); привілеї процесу; інформацію про володіння ресурсами та їх використання).

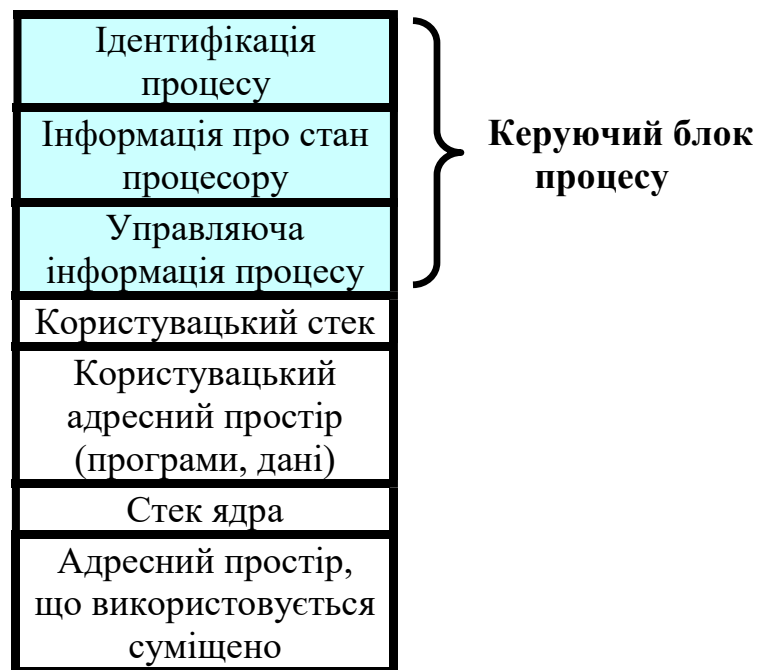


Рис. 2.4. Образ процесу

Процеси можна класифікувати так:

1. За приналежністю до центрального процесору (ЦП):
 - *внутрішні* (в операційних системах прийнято розрізняти не тільки час існування процесу, але й час його народження. Такою точкою відліку прийнято вважати ЦП. Процеси, виконані на ньому, називаються програмними або внутрішніми);
 - *зовнішні* (це процеси, розвиток яких проходить під контролем не ЦП, а інших процесорів).
2. За приналежністю до ОС:
 - *системні*;
 - *користувацькі*.
3. За генеалогічним типом:
 - *породжуючі* (це процеси, які можуть породжувати існування інших процесів);
 - *породжені* (процеси, які починають існувати в результаті існування іншого процесу).

4. За результативністю:

- *еквівалентні* (два процеси, які мають один і той самий результат обробки даних в одній і тій самій програмі на одному і тому самому або на різних процесорах);

- *тотожні* (якщо в кожному з еквівалентних процесів обробка даних проходить в одній і тій самій програмі, але траси не збігаються);

- *рівні* (під час збігання слідів тотожних процесів).

5. За динамічним принципом:

- *послідовні* (якщо інтервали двох процесів не пересікаються в часі);

- *паралельні* (якщо на певному інтервалі часу існують одночасно два процеси).

6. За пов'язаністю:

- *взаємопов'язані* (якщо між ними створюються зв'язки за допомогою системи управління процесів);

- *ізолювані*;

- *інформаційно-незалежні* (якщо два взаємопов'язані процеси використовують одні і ті самі ресурси, але не обмінюються між собою інформацією);

- *взаємодіючі* (якщо між двома процесами є інформаційні зв'язки).

2.1.4. Управління процесами

Створення процесів. Під час створення нового процесу ОС повинна виконати такі дії:

- 1) присвоїти новому процесу унікальний ідентифікатор, при цьому в таблицю процесів вноситься новий запис;

- 2) виділити простір для процесу – для образу процесу і для його керуючого блоку;

- 3) ініціалізувати керуючий блок процесу;

- 4) встановити необхідні зв'язки, наприклад включити новий процес у чергу готових процесів;

- 5) створити або розширити інші структури даних.

Перемикання процесів може відбутися в будь-який момент, коли управління передається операційній системі: у разі апаратного переривання, зокрема переривання від таймера, спрацьовування пастки (виключення), у разі системного виклику. Усі перераховані причини реалізуються *через механізм переривань*.

Дії ОС під час перемикання процесів:

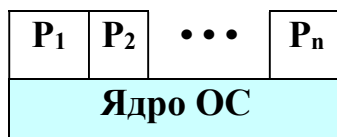
- 1) збереження контексту процесора (вмісту регістрів загального призначення і керуючих регістрів) у керуючому блоці процесу;
- 2) оновлення керуючого блоку виконується в цей момент процесу – потрібно змінити стан цього процесу і зберегти інформацію з обліку використання ним ресурсів;
- 3) переміщення керуючого блоку цього процесу у відповідну чергу (готових, заблокованих процесів та ін.);
- 4) вибір необхідного процесу для виконання;
- 5) оновлення керуючого блоку обраного процесу – необхідно встановити для нього стан виконання;
- 6) оновлення структур даних щодо управління пам'яттю;
- 7) відновлення контексту процесора для вибраного процесу.

Вважається, що ОС теж є програмою, яка виконується процесором, тобто виникає питання: чи є ОС процесом? Якщо так, то має здійснюватися певним способом управління таким процесом. Під час розв'язання цієї проблеми виникли різні підходи до розробки ОС:

- автономне ядро;
- ядро у складі користувацьких процесів;
- ОС на основі процесів.

Ядро поза процесами (автономне ядро) – використовується в багатьох ранніх ОС. Концепція процесу тут застосовується тільки до користувацьких програм, тоді як код ОС виконується як окремий об'єкт, що працює у привілейованому режимі (рис. 2.5[а]).

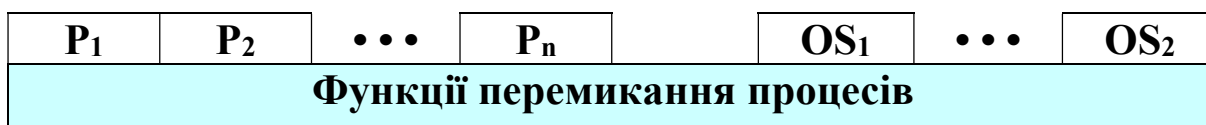
Ядро у складі користувацьких процесів – використовується в невеликих ЕОМ (у ПК і робочих станціях). Майже всі програми такої ОС виконуються в контексті користувача процесу (рис. 2.5[б]).



а) автономне ядро



б) функції операційної системи, що виконуються в користувацьких системах



в) функції операційної системи, що виконуються як окремі процеси

Рис. 2.5. Виконання коду ОС щодо процесів:

а) ядро поза процесами; б) ядро у складі користувацьких процесів; в) ОС на основі процесів

У разі виклику системних процедур, що працюють у режимі ядра, і повернення з них використовується окремий стек ядра. Код та дані ОС містяться у спільно використовуваному адресному просторі і доступні для використання всіма користувацькими процесами. Тому під час переривання, включаючи системні виклики, перемикавання процесів не відбувається, перемикається тільки режим роботи процесора (користувацький $\leftrightarrow$ ядро) в рамках одного і того самого процесу. При цьому в рамках одного і того самого процесу можуть виконуватися і користувацькі програми, і програми ОС. Програми ОС, виконувані в різних користувацьких процесах, є ідентичними.

ОС на основі процесів. На рисунку 2.5(в) показана структура ОС на основі процесів.

Переваги такого підходу:

- він сприяє розробці модульних ОС з простими міжмодульними інтерфейсами;

- деякі другорядні функції ОС зручно реалізувати у вигляді окремих процесів; ці функції може викликати тільки ОС, і такі процеси можуть чергуватися з користувацькими;
- в багатопроцесорних і багатокомп'ютерних обчислювальних системах окремі служби ОС можуть виконуватися на різних процесорах, що, відповідно, підвищує продуктивність системи.

2.1.5. Стратегії планування і диспетчеризація процесів

Планування – це системний процес, який здійснює установку користувацьких процесів у чергу та визначення атрибутів їх виконання в рамках використовуваної обчислювальної системи.

У загальному випадку стратегія планування визначає, які процеси повинні бути заплановані на виконання з метою досягнення результатів задачі, яка вирішується.

Стратегії можуть бути такими:

- закінчувати обчислення в тому самому порядку, в якому вони були розпочаті;
- перевагу надавати більш коротким процесам;
- надавати всім процесам однакові послуги, зокрема й однаковий час очікування.

Розрізняють три типи планування:

- *короткострокове планування* – диспетчеризація – рішення про додавання процесу в пул процесів, що виконуються;
- *середньострокове планування* – рішення про додавання процесу до процесів, що повністю або частково розміщені в оперативній пам'яті;
- *довгострокове планування* – рішення про те, який з доступних процесів буде виконуватися процесором (полягає у виборі таких обчислювальних процесів, які менше всього б конкурували між собою в процесі досягнення мети обчислень).

Системний процес, який здійснює планування, може бути реалізований двома способами:

- *цілісний планувальник* – це програмний модуль, який є частиною операційної системи, його робота ініціюється перериваннями і він виконується як окремий системний процес;

- *розподілений планувальник* – програмний модуль планувальника або його частина записується в кожний процес у разі його завантаження в оперативну пам'ять.

У загальному випадку планувальник здійснює такі функції:

- 1) запис копії процесу в пам'ять обчислювальної системи;
- 2) аналіз атрибутів під час виконання обчислювального процесу та формування набору робочих атрибутів залежно від характеристик обчислювальної системи;

- 3) запис процесу в чергу процесів у випадку, якщо є вільне місце в черзі, або перевід процесу в пасивний стан у випадку, якщо всі елементи черги зайняті;

- 4) періодичний перегляд черги процесів, запис або знищення процесів у черзі згідно з часом функціонування процесів.

Відома велика кількість правил (дисциплін диспетчеризації), відповідно до яких формується список (черга) готових до виконання задач.

Розрізняють два великих класи дисциплін диспетчеризації: безпріоритетні і пріоритетні.

За безпріоритетної організації задачі або процеси вибираються згідно з певним, раніше установленим порядком без врахування важливості цих процесів та необхідного часу обслуговування.

З реалізації пріоритетних дисциплін певним окремим задачам чи процесам надається виключне право або набір прав для установки такого процесу в чергу.

На рисунку 2.6 представлена структура дисциплін диспетчеризації.

Далі опишемо принципи роботи деяких дисциплін диспетчеризації:

1. *Дисципліна FIFO (first input first output)* – є реалізацією безпріоритетної черги. Якщо в черзі звільняється елемент і він не останній, то всі після нього посуваються вперед, і в останній вільний елемент черги записується ім'я нового процесу. Першим з черги вибирається елемент, який стоїть на початку черги.

2. *Дисципліна LIFO (last input first output)* – першим з черги обирається елемент, який прийшов останнім.



Рис. 2.6. Структура диспетчеризації

3. *Дисципліна FCFS (first come first server)* – подібна до FIFO. Ця дисципліна враховує перебування процесів у станах блокування, наприклад щодо операцій введення / виведення. Ті, які були заблоковані в процесі роботи після переходу в стан готовності, поступають у чергу готовності перед новими задачами, які ще не обслуговувались. За такої дисципліни організовуються дві черги: черга готових до обслуговування процесів і черга нових процесів (рис. 2.7).

Ця дисципліна не потребує втручання ззовні у хід обчислювального процесу. Не відбувається розподіл процесорного часу.

Переваги: простота реалізації та невеликий розхід системних ресурсів для організації черги задачі. Однак у разі збільшення завантаженості ОС збільшується середній час обслуговування задачі, коли «короткі» завдання (які потребують невеликих затрат машинного часу) повинні очікувати такий самий час, що й довгі завдання.

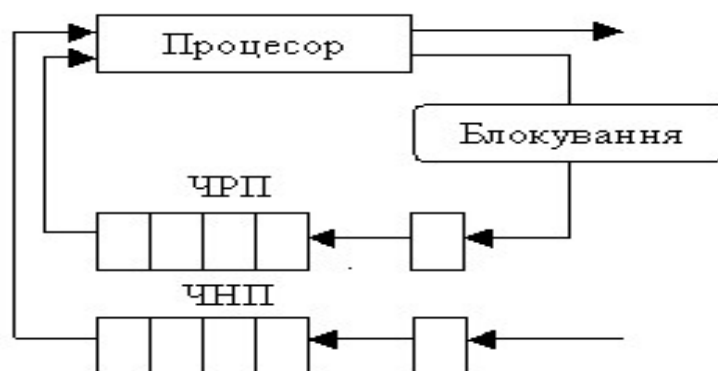


Рис. 2.7. Дисципліна FCFS (ЧНП – черга нових процесів; ЧРП – черга робочих процесів)

4. *Дисципліна SJN (short jump next)* – згідно з цією дисципліною ОС вимагає, щоб для кожного процесу була відома оцінка необхідних обчислювальних ресурсів. Для постановки задачі в чергу диспетчер оцінює необхідний час виконання задачі і ставить задачу перед більш довгою задачею. За цією дисципліною існує одна черга процесів, і завдання, що були заблоковані, знову поступають у кінець черги, як і нові завдання. Це приводить до того, що завдання, які потребують мало часу, очікують процесу, як і довгі процеси.

5. *Дисципліна SRT (short remain time)* – розроблена з метою забезпечення більш якісного обслуговування коротких завдань. Виконує спочатку ті процеси, час завершення виконання яких найменший.

6. *Дисципліна RR (round robin)* – кожна задача отримує порцію часу (квант часу). Після закінчення цього кванту часу задача знімається з виконання на процесорі і він передається наступній задачі. Задача, яка була знята з черги, записується в кінець черги задач, які готові до виконання. Величина кванту часу, як правило, обирається як середнє значення між достатнім часом реакції системи на запити користувачів та процесорним часом, який необхідний для перемикання між задачами.

2.1.6. Алгоритми в диспетчеризації з витісненням та без витіснення

Диспетчеризація без розподілу процесорного часу (багато-задачність без витіснення) – це такий спосіб диспетчеризації, при якому активний процес виконується доти, доки він з власної ініціативи не передасть управління диспетчеру задач для вибору іншого, готового до виконання процесу. До них належать такі дисципліни: FCFS, SJN, SRT.

Диспетчеризація з розподілом процесорного часу (багато-задачність з витісненням) – це такий спосіб, де рішення про перемикання з однієї задачі на іншу приймається диспетчером задач, а не власною активною задачею. Процес, що виконується, може бути перерваним і переведений операційною системою в стан готовності до виконання. Рішення про витіснення може прийматися під час запуску нового процесу з переривання або періодично згідно з перериваннями від таймера. До таких дисциплін належать RR та інші, реалізовані на її основі.

Будь-які процеси мають бути гарантовано обслуговані операційною системою. Гарантоване обслуговування може бути досягнуто трьома способами:

- 1) ОС виділяє мінімальну частину процесорного часу деякому класу процесів у випадку, коли хоча б один з них готовий до виконання;
- 2) ОС виділяє мінімальну долю процесорного часу певному конкретному процесу, якщо він готовий до виконання;
- 3) ОС виділяє стільки часу певному процесу, щоб він міг виконати своє обчислення в певний термін.

Для порівняння алгоритмів диспетчеризації можуть бути використані такі критерії:

- 1) *використання (завантаження) процесору* – відсоток часу, впродовж якого процесор зайнятий;
- 2) *пропускна здатність процесора* – це кількість процесів, яка виконується процесором за одиницю часу;

3) *час обороту* – інтервал часу від моменту появи процесу у вхідній черзі до моменту завершення процесу. Цей час обороту включає в себе час очікування у вхідній черзі, час очікування у черзі готовності, час готовності у чергах до периферійних пристроїв, час виконання на процесорі та час введення / виведення;

4) *час очікування* – це сумарний час перебування процесу в черзі очікування готових процесів;

5) *час відповіді* – це час від моменту поступлення процесу на вхідну чергу до моменту першого звернення процесу на введення / виведення даних.

Як відомо, під час виконання операційною системою багатьох процесів може відбуватися зниження продуктивності обчислювальної системи.

Існують певні методи підвищення продуктивності системи:

- *сумісне планування*, при якому всі потоки однієї задачі одночасно обираються для виконання процесором (у випадку наявності мультипроцесорної системи) й одночасно знімаються з виконання. У цьому випадку зменшується час перемикання між задачами;

- *планування*, при якому задачі, що перебувають у критичній сфері, не перериваються, а закінчують виконання критичної секції. Задачі, які чекають на виконання критичної секції коду, не виконують її, доки не завершиться переривання.

У разі виконання програм, які реалізують функції контролю чи управління в системах реального часу, може виникнути ситуація, коли одна або декілька задач не можуть бути вирішені упродовж довгого проміжку часу через значну завантаженість обчислювальної системи. У цьому випадку втрати, які пов'язані з невиконанням цих задач, можуть бути більшими, ніж втрати від невиконання задач з більш високим пріоритетом. Тому іноді виникає необхідність динамічної зміни пріоритету задачі в процесі її виконання. Це дозволяє реалізувати більш швидку реакцію на запити користувачів і гарантувати виконання будь-яких запитів.

Операційна система може змінити пріоритет задачі за допомогою:

1) підвищення пріоритету активної задачі: під час завантаження задачі на виконання її пріоритет автоматично збільшується і при цьому знижується час реакції цієї активної задачі на дії користувачів порівняно з іншими фоновими задачами;

2) підвищення пріоритету операції введення / виведення: після завершення операції введення / виведення задача отримує найвищий пріоритет у певному класі задач, що дозволяє більш швидко закінчити всі незавершені операції введення / виведення;

3) підвищення пріоритету «забутих» задач: якщо задача не отримує часу процесора упродовж певного відрізка часу, то диспетчер задач тимчасово присвоює їй більш високий пріоритет, який не перевищує певної заданої межі, і таким способом можливо перемикається на ці «забуті» задачі більш швидко. Після виконання такої задачі за певний квант часу її пріоритет знижується до попереднього значення.

2.2. Взаємодія процесів

2.2.1. Проблема взаємного виключення і способи її вирішення

Два процеси називаються *паралельними (concurrent)*, якщо їх виконання може перекриватися в часі, тобто, наприклад, другий процес починається раніше, ніж завершується перший. Два процеси називаються *послідовними (serial)*, якщо вони не є паралельними.

У мультипроцесорній системі досягається *фізична паралельність*. Якщо виконання декількох процесів чергується в єдиному процесорі, то досягається *логічна паралельність*.

Можна виділити три види взаємодії процесів:

- *обмін інформацією* – якщо процеси безпосередньо проінформовані про наявність один одного;
- *конкуренція за доступ до ресурсів* – якщо процеси не проінформовані про наявність один одного; ОС повинна регулювати такі звернення;

- *узгодження дій процесів* – якщо процеси непрямо обізнані один про одний (наприклад, процес А генерує дані, а процес В виводить їх на друк).

При цьому паралельні процеси взаємодіють за допомогою або *механізму поділюваних змінних*, або *механізму передачі повідомлень*.

Під час роботи паралельних процесів як в однопроцесорних, так і в багатопроцесорних системах неможливо передбачити відносну швидкість їх роботи, і тому виникають проблеми у разі взаємодії таких процесів.

1. *Стан гонок* (змагань), коли два процеси використовують одну і ту саму глобальну змінну й обидва виконують читання і запис цієї змінної. Критичним при цьому виявляється порядок читання і запис цієї змінної різними процесами.

2. *Блокування та взаємоблокування під час доступу до ресурсів*¹. Процес А може зажадати й одержати контроль над деяким пристроєм введення / виведення (ПВВ), після чого тимчасово призупинити роботу. При цьому небажано, щоб ОС блокувала цей ПВВ і не дозволяла іншим процесам використовувати його, оскільки це може призвести до взаємоблокування.

3. *Програмні помилки*, які важко виявити, оскільки результат роботи програми перестає бути детермінованим і відтворюваним.

Приклад стану гонок. Нехай у нас є процедура, що виконує читання символу з клавіатури і його відображення на екрані, яка міститься в пам'яті, що розділяється і використовується всіма процесами:

```
void echo ()  
{  
  chin = getchar (); chout = chin; putchar (chout);  
}
```

¹ Питання взаємоблокування винесено в окремий підрозділ 2.4. Взаємне блокування.

Розглянемо таку послідовність подій:

- 1) процес А виконує оператор `ch = getchar ()` – введено символ «х»;
- 2) процес В виконує процедуру `echo` до кінця – введено і надруковано символ «у»;
- 3) процес А продовжує своє виконання, але при цьому буде виведений уже символ «у», а не «х», тобто «у» буде виведений двічі, а «х» – втрачений.

Така проблема не виникатиме, якщо в кожен момент часу тільки один процес буде входити в процедуру *echo*, і ця процедура обов'язково повинна бути повністю виконана процесом, що увійшов до неї до того, як стане можливим її виконання іншим процесом.

Та сама проблема залишається і в багатопроцесорній системі.

Для уникнення проблем типу гонок, блокувань та взаємоблокувань у концепції процесів існує *механізм взаємного виключення* (mutual exclusion).

Взаємне виключення – механізм, який гарантує, що в будь-який момент часу тільки один процес виконує деяку визначену послідовність дій, і тим самим виключає можливість роботи іншого процесу.

Критичний ресурс (КР) – ресурс, до якого в кожний момент часу можливий доступ тільки одного процесу. Доступ до такого ресурсу здійснюється в **критичній секції (КС)** – частини коду, яка в будь-який момент часу може виконуватися тільки одним процесом.

Але взаємне виключення може призвести до двох проблем:

- 1) *взаємного блокування (deadlock)*;
- 2) *голодування (starvation)*.

Розглянемо, як виникає проблема голодування. Нехай є процеси А, В і С, кожному з яких періодично потрібен доступ до ресурсу Р. Нехай процес А використовує ресурс Р, а процеси В і С – очікують. Потім процес В використовує ресурс Р, після цього знову процес А використовує ресурс Р тощо. Тобто процес С може ніколи не отримати доступ до ресурсу Р, незважаючи на те,

що ніякого взаємного блокування немає. Така ситуація і називається *голодуванням*.

Отже, щоб уникнути гонок, маємо уникати перебування двох процесів одночасно в критичних секціях. Але цієї вимоги ще недостатньо. Для правильної спільної роботи паралельних процесів необхідно виконання шести умов для досягнення взаємного виключення:

- 1) два процеси не повинні одночасно перебувати в критичних секціях;
- 2) у програмі не повинно бути закладено ніяких початкових припущень про швидкість або кількість процесорів;
- 3) процес, що перебуває поза критичною секцією, не повинен блокувати інший процес;
- 4) повинна бути неможлива ситуація, коли процес постійно чекає попадання в критичну секцію (тобто не повинні з'являтися взаємоблокування і голодування);
- 5) коли в КС немає жодного процесу, будь-який процес, що запитав можливість входу в цю КС, повинен негайно отримати доступ до неї;
- 6) процес залишається в КС лише упродовж обмеженого часу.

Існують такі підходи до досягнення взаємного виключення:

- *програмний* – у цьому випадку сам процес є «відповідальним» за досягнення взаємного виключення, тобто процес (як системний, так і користувацький) повинен координувати свої дії з іншими процесами для досягнення взаємного виключення без підтримки з боку ОС або середовища програмування;
- *апаратний* – з використанням спеціальних машинних команд;
- *примітиви синхронізації* – семафори, монітори та інші подібні засоби.

2.2.2. Апаратні способи досягнення взаємного виключення

Існують такі апаратні способи досягнення взаємного виключення:

- *заборона переривань;*
- *спеціальні команди, що виконуються атомарно.*

Під час *заборони переривань* процесор перемикається на інший процес тільки щодо переривання. Відповідно, процес забороняє переривання перед входом у критичну секцію і дозволяє переривання відразу після виходу з критичної секції. Взаємне виключення при цьому досягається, тому що процес не переривається всередині критичної секції, і тому інші процеси не можуть увійти у свої критичні секції.

У такого підходу є певні недоліки: по-перше, він може бути застосований тільки в однопроцесорних системах, по-друге, важливі події введення / виведення можуть бути не оброблені і, по-третє, у разі краху процесу всередині критичної секції може відбутися порушення роботи ОС.

Спеціальні команди, що виконуються атомарно. Під атомарними операціями маються на увазі такі команди зміни даних в оперативній пам'яті, які виконуються процесором з повним блокуванням доступу до пам'яті, тобто жоден інший процесор або зовнішній пристрій не може отримати доступ до комірок пам'яті, доки процесор-монополіст не завершить операцію з нею.

Розглянемо реалізацію взаємного виключення за допомогою атомарної команди «перевірити і встановити» (TestAndSet). Ця команда виконує атомарно дві операції: читання з комірки пам'яті і запис у неї значення «зайнято».

Нехай нульове значення змінної *lock* відповідає умові, що критична секція вільна, а значення *lock*, відмінне від нуля, відповідає зайнятості критичної секції. Тоді перед входом у критичну секцію процес повинен у циклі перевіряти, чи звільнилася критична секція:

```
while (TestAndSet (& lock));  
/* Критична секція */
```

Відповідно, після виходу з критичної секції процес повинен встановити змінну в значення «вільно»:

$lock = 0$.

2.2.3. Програмні способи досягнення взаємного виключення

Розрізняють такі програмні способи досягнення взаємного виключення:

- змінні блокування;
- алгоритм Деккера;
- алгоритм Петерсона.

Змінні блокування. Процес для здійснення входу в критичну секцію безперервно перевіряє значення деякої змінної, щоб визначити, чи є вільним ресурс, що розділяється. Ця змінна має назву «змінна блокування» і зберігає стан критичної секції. Початкове значення змінної блокування дорівнює 0. Якщо процес хоче увійти в критичну секцію, то він попередньо зчитує значення змінної блокування, і якщо воно дорівнює 0, то процес змінює його на 1 і входить у критичну секцію. Якщо ж значення цієї змінної вже дорівнює 1, то процес чекає, доки це значення не зміниться на 0. Отже, 0 означає, що жодного процесу в критичній секції немає, а 1 – певний процес уже перебуває в критичній секції.

```
while (busy);
busy = 1;
/* Критична секція */
.....
busy = 0;
/* Решта коду */
.....
```

При такому підході виникають певні проблеми, а саме:

1) ті самі проблеми, що і під час організації доступу до загальних змінних;

2) якщо після виходу з критичної секції процес з будь-якої причини не встановить значення змінної блокування, що дорів-

нює 0, то інші процеси не зможуть увійти в критичну секцію, порушується умова про те, що процес, який перебуває поза критичною секцією, не повинен блокувати інший процес;

3) проблема непродуктивної витрати часу процесору.

Активне очікування (busy waiting) – постійна перевірка змінної в очікуванні деякого значення. *Спін-блокування* – блокування, що використовує активне очікування.

Алгоритм Деккера. Голландський математик Т. Деккер був першим, хто розробив програмне рішення проблеми взаємного виключення (1970). Коли процес $P0$ намагається увійти в критичну секцію, він встановлює свій прапор $flag[0]$ таким, що дорівнює *true*, а потім перевіряє стан прапора іншого процесу $P1$. Якщо його прапор дорівнює *false*, то процес $P0$ може негайно увійти в критичну секцію, інакше $P0$ звертається до змінної *turn*.

Якщо $turn = 0$, то це означає, що в цей момент черга процесу $P0$ на вхід у критичну секцію, і тоді $P0$ періодично перевіряє стан прапора процесу $P1$. Цей процес зі свого боку в певний момент часу виявляє, що нині не його черга для входу в критичну секцію, і встановлює свій прапор, який дорівнює *false*, даючи можливість процесу $P0$ увійти в критичну секцію.

А після того, як $P0$ вийде з критичної секції, він встановить свій прапор, який дорівнює *false*, для звільнення критичної секції і присвоїть змінній *turn* значення 1 для передачі прав на вхід у критичну секцію процесу $P1$:

$P0$	$P1$
<pre> flag[0] = true; while (flag[1]) { if(turn == 1) { flag[0] = false; while(turn == 1) /* очікувати */; flag[0] = true; } } /* критична секція */ turn = 1; flag[0] = false; </pre>	<pre> flag[1] = true; while (flag[0]) { if (turn == 2) { flag[0] = false; while(turn == 1) /* очікувати */; flag[1] = true; } } /* критична секція */ turn = 0; flag[1] = false; </pre>

Алгоритм Пітерсеа. Г. Пітерсон розробив набагато більш простий алгоритм взаємного виключення (1981 р.). З цього моменту алгоритм Деккера вважається застарілим.

```
int turn; /* 0 або 1 - чия зараз черга? */  
int interested [2];  
/* Всі змінні спочатку рівні 0 */  
  
void enter_cs (int proc)  
{  
    int other = 1 - proc;  
    interested [proc] = true;  
    turn = proc;  
    while (turn == proc && interested [other]);  
}  
  
void leave_cs (int proc)  
{  
  
    interested [proc] = false;  
}
```

Перед входом у критичну секцію процес викликає функцію *enter_cs* зі своїм номером (0 або 1) як параметр. Після виходу з КС процес викликає функцію *leave_cs*, щоб позначити свій вихід і тим самим дозволити іншому процесу вхід у КС.

Початково обидва процеси перебувають поза КС. Коли процес 0 викликає функцію *enter_cs*, ця функція повертає керування тому, що процес 1 не зацікавлений у попаданні в КС. Тепер якщо процес 1 теж викличе функцію *enter_cs*, то йому доведеться чекати, коли *interested [0]* дорівнюватиме *false*, а це відбудеться тільки в той момент, коли процес 0 викличе функцію *leave_cs*.

Нехай тепер обидва процеси намагаються викликати функцію *enter_cs* практично одночасно. Тоді обидва вони збережуть свої номери в змінну *turn*. При цьому в ній збережеться номер того процесу, який був другим, а попередній номер буде загублений.

Нехай *другим* був процес 1 і, відповідно, $turn = 1$. У цьому випадку, коли обидва процеси дійдуть до оператора *while*, процес 0 увійде в КС, а процес 1 залишиться в циклі і чекатиме, доки процес 0 вийде з КС.

2.3. Примітиви синхронізації

2.3.1. Семафори

Семафор – це спеціальна змінна, яка має значення і пов'язану з ним чергу. Над семафором визначено три операції:

- 1) семафор може бути ініціалізований негативним значенням;
- 2) операція *wait* зменшує значення семафора; якщо це значення стає негативним, то процес, що виконує операцію *wait*, блокується;
- 3) операція *signal* збільшує значення семафора; якщо це значення не позитивне, то заблокований операцією *wait* процес розблокується.

Існує також більш обмежена версія семафора – *бінарний семафор*, що приймає тільки значення 0 або 1.

Для зберігання процесів, що очікують семафори, використовується *черга*. Якщо вона обслуговується за принципом FIFO, то такий семафор називається *сильним семафором*, інакше – *слабким семафором*.

Варто зазначити, що семафори забезпечують рішення для всіх видів проблем синхронізації, але їм також властиві недоліки:

- 1) процес, що використовує семафор, повинен бути обізнаний про інші процеси, що використовують цей самий семафор, оскільки операції над семафорами у всіх взаємодіючих процесах повинні бути чітко скоординовані;
- 2) операції над семафорами у всіх процесах повинні бути ретельно налагоджені, оскільки пропуск однієї з таких операцій може призвести до неспроможності (порушення цілісності ресурсу, що розділяється) або до взаємного блокування;

3) програми, що використовують семафори, досить важко перевіряти на коректність.

Наведемо приклад застосування семафорів у разі взаємного виключення. Розглянемо рішення задачі взаємовиключення з використанням семафора s . Нехай є n процесів. У кожному з цих процесів перед входом у критичну секцію виконується виклик $wait(s)$. Якщо значення s при цьому стає негативним, то відповідний процес призупиняється. Якщо ж це значення дорівнює 1, то воно зменшується до нуля, а процес негайно входить у критичну секцію. І оскільки s вже більше не є позитивним, жоден інший процес не може увійти в критичну секцію.

```
semaphore s = 1;  
.....  
wait (s);  
/* Критична секція */  
signal (s);  
/* Решта коду */
```

Семафор ініціалізується значенням 1. Отже, перший процес, що виконує виклик $wait(s)$, може негайно потрапити в критичну секцію, встановлюючи при цьому значення семафора, що дорівнює 0. Будь-який інший процес у разі спроби увійти в критичну секцію виявить, що вона зайнята. Відповідно, відбудеться блокування такого процесу, а значення семафора буде зменшено до -1 . Намагатися увійти в критичну секцію може будь-яка кількість процесів, і кожна така неуспішна спроба зменшує значення семафора. Після того, як процес, який увійшов у критичну секцію першим, покине його, значення s збільшується, а один із заблокованих процесів (якщо такі є) віддаляється з черги семафора й активізується. Отже, як тільки планувальник ОС надасть цьому процесу можливість виконання, процес одразу зможе увійти в критичну секцію.

Сигналізуючий семафор – це семафор з нульовим початковим значенням. Процес сигналізує про подію, виконуючи операцію

signal (s), а всі інші процеси очікують події, виконуючи операцію *wait (s)*.

2.3.2. М'ютекси

М'ютекс – це спрощена версія семафора: змінна, яка може перебувати в одному з двох станів – блокованому і неблокованому.

Значення м'ютекса встановлюється за допомогою двох процедур. Якщо процес (або потік) збирається увійти в критичну секцію, то він викликає процедуру *mutex_lock*. Якщо м'ютекс незаблокований, тобто вхід у критичну секцію дозволений, то запит виконується, а процес, що викликає, входить у критичну секцію. Якщо ж м'ютекс заблокований, то процес блокується доти, доки інший процес, що перебуває в критичній секції, не вийде з неї, викликавши процедуру *mutex_unlock*. Якщо при цьому м'ютекс блокував декілька процесів, то з них обирається один.

2.3.3. Монітори

Щоб спростити написання програм, був запропонований механізм синхронізації більш високого рівня.

Монітор – це набір процедур, змінних та інших структур даних, об'єднаних в особливий модуль. Він являє собою високо-рівневу конструкцію мови програмування, яка забезпечує функціональність, еквівалентну до функціональності семафорів, але при цьому ним набагато легше управляти. Монітори реалізовані в мовах програмування Concurrent Pascal, Pascal-Plus, Modula-2, -3, Java.

Монітор являє собою програмний модуль, який складається з послідовності, що ініціалізується однією або декількома процедурами, і локальних даних. *Основні характеристики монітора:*

1) локальні змінні монітора доступні тільки його процедурам; зовнішні процедури доступу до локальних даних монітора не мають;

2) процес входить у монітор шляхом виклику однієї з його процедур;

3) у моніторі в певний момент часу може виконуватися тільки один процес; будь-який інший процес, що викликав монітор, буде припинений в очікуванні доступності монітора.

Перші дві характеристики нагадують нам об'єкти в об'єктно-орієнтованому програмуванні; дотримання ж третьої умови дозволяє монітору забезпечити взаємовиключення. Дані монітора доступні в деякий конкретний момент тільки одному процесу, отже, захистити спільно використовувані дані можна, помістивши їх у монітор. Якщо дані в моніторі являють собою якийсь ресурс, то монітор забезпечує взаємовиключення у разі звернення до цього ресурсу.

Для широкого застосування в паралельних обчисленнях монітори повинні включати в себе інструмент синхронізації. Припустимо, наприклад, що якийсь процес використовує монітор і, перебуваючи в цьому моніторі, він повинен бути припинений до виконання певної умови. При цьому потрібен механізм, який не тільки призупиняє процес, але і звільняє монітор, дозволяючи увійти в нього іншому процесу. Пізніше, коли умова виявиться виконаною, а монітор доступним, припинений процес зможе продовжити свою роботу з того місця, де він був призупинений.

Монітор підтримує синхронізацію за допомогою *змінних умови*, що розташовуються в моніторі і доступних тільки в моніторі. Працювати з цими змінними можуть дві функції:

1) *swait (c)* – призупиняє виконання процесу, що викликає за умовою *c*; монітор при цьому доступний для використання іншим процесом;

2) *csignal (c)* – відновлює виконання процесу, припиненого викликом *swait (c)*; якщо є декілька таких процесів, то обирається один з них, а якщо таких процесів немає, то ця функція нічого не виконує.

2.4. Взаємні блокування

Взаємне блокування – це блокування групи процесів, що очікують подію, яка може ніколи не настати, оскільки цю подію може викликати тільки інший процес з тієї саме групи. Взаємоблокування зазвичай виникають під час доступу процесів до ресурсів.

Ресурс – це будь-який об'єкт, який може запитуватися й очікуватися процесом; ресурс може складатися з будь-якої кількості ідентичних одиниць, і процес може запитувати будь-яку кількість одиниць ресурсу.

Існують дві *основні категорії ресурсів*:

1. *Повторно використовувані ресурси (reusable)* не виснажуються під час використання (можуть використовуватися знову). Вони мають фіксовану кількість одиниць, що не може створюватися або знищуватися. Процес, що запитує такий ресурс, утримує його у разі використання, потім звільняє, а після цього ресурс може бути виділений іншим процесам. Приклади повторно використовуваних ресурсів: процесор, пристрій введення / виведення, основна та вторинна пам'ять, структури даних (файли).

2. *Ресурси, що витрачаються (consumable)*, можуть бути створені (вироблені) або знищені (спожиті). Якщо процес використав такий ресурс, то останній припиняє своє існування («витрачається»). Зазвичай обмеження на кількість витрачених ресурсів певного типу відсутні, і незаблокований процес-виробник може створити будь-яку кількість таких ресурсів. Приклади ресурсів, що витрачаються: переривання, сигнали, повідомлення, інформація в буферах введення/виведення.

Є два режими доступу до ресурсу:

- *монопольний (exclusive)*;
- *розділюваний (shared)*.

Під час *монопольного (exclusive) режиму* доступ до ресурсу в кожний момент часу може отримати тільки один процес. У разі *розділюваного (shared) режиму* доступ до ресурсу одночасно може отримати будь-яка кількість процесів.

Ці режими несумісні, тобто до ресурсу не може бути реалізований доступ одночасно в монопольному і розділюваному режимах. Якщо ж існує багато одиниць (копій) ресурсу, то доступ до одних з них може здійснюватися в розділюваному режимі для декількох процесів, а доступ до інших – у монопольному режимі. Очевидно, що два цих режими доступу мають сенс тільки для повторно використовуваних ресурсів, оскільки споживані ресурси знищуються відразу після використання.

Приклад 1. Взаємне блокування з повторно використовуваними ресурсами.

<i>Процес P1</i>
Запит ресурсу А
Запит ресурсу В

<i>Процес P2</i>
Запит ресурсу В
Запит ресурсу А

Коли обидва ці процеси дійдуть до другого запиту, виникне взаємне блокування. Один зі способів уникнути його – це накласти системні обмеження на порядок запиту ресурсів.

Приклад 2. Взаємне блокування з повторно використовуваними ресурсами.

Нехай нам доступно 200 Кб ОП і виконується така послідовність запитів:

<i>Процес P1</i>
Запит 80 Кб
Запит 60 Кб

<i>Процес P2</i>
Запит 70 Кб
Запит 80 Кб

Якщо обидва ці процеси дійдуть до другого запиту, то виникне взаємне блокування, незважаючи на те, що в системі досить пам'яті для виконання кожного з процесів окремо.

Приклад 3. Взаємне блокування з ресурсами, що витрачаються.

Нехай два процеси намагаються одержати повідомлення від іншого процесу, а потім відправити йому повідомлення.

Процес P1	Процес P2
$m1 = Receive (P2);$	$m1 = Receive (P1)$
$Send (P2, m2);$	$Send (P1, m2);$

Взаємне блокування при цьому виникне, якщо операція *Receive* є блокуючою.

Умови виникнення взаємного блокування:

- 1) *взаємне виключення* – одночасно використовувати ресурс може тільки один процес;
- 2) *утримання та очікування* – процес може утримувати виділені ресурси під час очікування інших ресурсів;
- 3) *відсутність перерозподілу* – ресурс не може бути примусово забраний у процесу, що його утримує;
- 4) *циклічне очікування* – існує замкнутий ланцюг процесів, кожний з яких утримує як мінімум один ресурс, необхідний процесу, наступного у ланцюзі після цього процесу.

Моделювання взаємних блокувань

Узагальнений ресурсний граф – це двочастковий спрямований граф з непересічними множинами вузлів процесів і ресурсів.

Вектор доступності – невід’ємний вектор $(r_1, r_2, \dots, r_m)$, що позначає кількість одиниць ресурсів, доступних в будь-якому стані.

Кожен процес на узагальненому ресурсному графі позначається кружечком, а ресурс – прямокутником, де ресурс, що використовується, – це прямокутник з подвійною рамкою. Кружечки всередині прямокутників позначають кількості одиниць цього типу ресурсу. Дуга, спрямована від процесу P до ресурсу R, називається *дугою запиту*. Вона означає, що процесу P призначено одну одиницю ресурсу R. Дуга, спрямована від повторно використовуваного ресурсу R до процесу P, називається *дугою призначення*. Вона означає, що процесу P призначена одна одиниця ресурсу R. Дуга, спрямована від ресурсу R, що використовується, до процесу P, називається *дугою виробника*. Вона означає, що процес P є виробником ресурсу R.

Приклад (рис. 2.8). Процеси P_1 і P_2 утримують по одній одиниці ресурсу R_1 , при цьому процес P_2 є виробником споживаного ресурсу R_2 . Доступно по одній одиниці кожного ресурсу. Процес P_1 активний, а P_2 – заблокований і очікує призначення одиниці ресурсу R_1 . Процес P_i є заблокованим у будь-якому стані тоді і тільки тоді, коли для деякого ресурсу R_j кількість дуг запиту більше r_j .

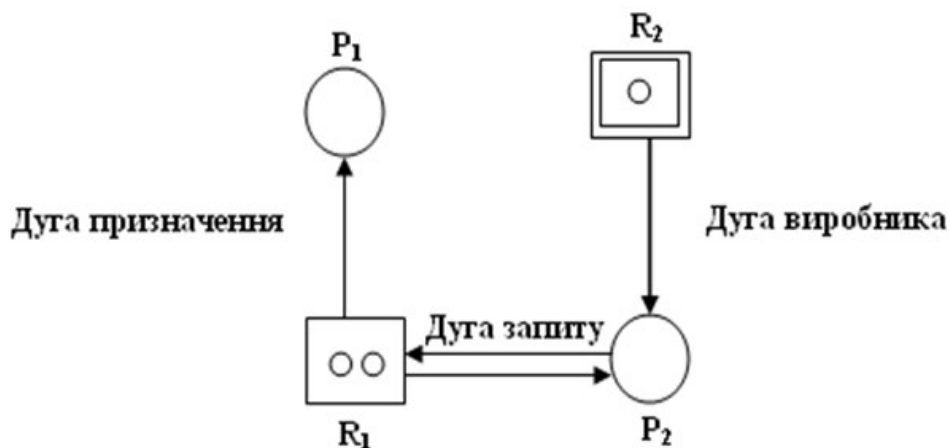


Рис. 2.8. Приклад графа ресурсів

Цикл у графі означає наявність взаємоблокування, циклічно включає в себе процеси і ресурси (припускаємо, що в системі є по одному ресурсу кожного виду).

Нехай існує три процеси: А, В, С, і три ресурси: R, S, Т, а також є така послідовність запитів і повернень ресурсу:

Процес А		Процес В		Процес С
запит R		запит S		запит Т
запит S		запит Т		запит R
звільнення R		звільнення S		звільнення Т
звільнення S		звільнення Т		звільнення R

ОС може запустити будь-який незаблокований процес будь-коли. Нехай запити ресурсів відбуваються у такому порядку:

- процес А запрошує R;
- В запрошує S;
- С запрошує E;

- А запрошує S;
- В запрошує T;
- С запрошує R.

Якщо ці шість запитів будуть здійснюватися в зазначеній послідовності, то отримаємо граф, показаний на рисунку 2.9, тобто відбудеться взаємне блокування.

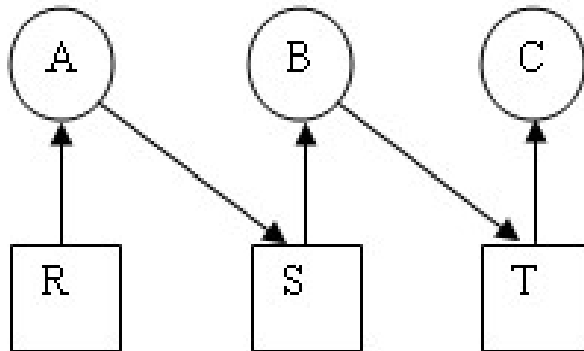


Рис. 2.9. Взаємне блокування

Однак ОС не зобов'язана запускати процеси в якомусь особливому порядку. Зокрема, якщо виконання будь-якого запиту спричиняє глухий кут, ОС може призупинити відповідний процес без задоволення запиту (тобто не виконуючи план цього процесу) доти, доки це безпечно. У нашому випадку ОС могла б призупинити процес В, якби вона «знала» про виникнення взаємоблокування, і тим самим виключити це взаємоблокування.

Графи ресурсів є інструментом, що дозволяє побачити, чи може задана послідовність дій стати причиною взаємоблокування. Для цього після кожного кроку граф перевіряється на наявність у ньому циклів.

Важливим питанням є виявлення взаємних блокувань. Це можливо за наявності або одного ресурсу кожного типу, або декількох ресурсів кожного типу.

Виявлення взаємних блокувань за наявності одного ресурсу кожного типу. У такій ситуації необхідною і достатньою умовою взаємоблокування є наявність циклу у графі ресурсів.

Виявлення взаємних блокувань за наявності декількох ресурсів кожного типу. Нехай n – кількість процесів $P_1, P_2, \dots, P_m$;

m – кількість класів ресурсів $R_1, R_2, \dots, R_m$;

E_i – кількість одиниць ресурсу класу;

E – вектор існуючих ресурсів, що містить загальну кількість наявних у системі одиниць кожного ресурсу;

A – вектор доступних ресурсів, елемент якого A_i є кількість одиниць ресурсу R_i , що доступні в цей момент;

C – матриця поточного розподілу, елемент якої C_{ij} є кількість одиниць ресурсу R_j , використовуване процесом P_i ;

R – матриця запитів, елемент якої R_{ij} є кількість одиниць ресурсу R_j , яку запитує процес P_j .

Останні чотири структури показані на рисунку 2.10.

<i>Вектор існуючих ресурсів</i>	<i>Матриця поточного розподілу</i>	<i>Матриця запитів</i>
$E = (E_1 \ E_2 \ E_3 \dots E_m)$		
<i>Вектор доступних ресурсів</i>	$C = \begin{bmatrix} C_{11} & C_{12} & C_{13} & \dots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \dots & C_{2m} \\ \dots & \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & C_{n3} & \dots & C_{nm} \end{bmatrix}$	$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} & \dots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \dots & R_{2m} \\ \dots & \dots & \dots & \dots & \dots \\ R_{n1} & R_{n2} & R_{n3} & \dots & R_{nm} \end{bmatrix}$
$A = (A_1 \ A_2 \ A_3 \dots A_m)$		

Рис. 2.10. Структури даних, що необхідні для алгоритму виявлення взаємних блокувань

Нехай у нас є алгоритм, який ставить на процесах відмітку, і вона є ознакою того, що ці процеси можуть закінчити роботу, і, отже, не перебувають у ситуації взаємного блокування. Тоді після завершення такого алгоритму будь-який немаркований процес перебуває в глухому куті.

Подібний алгоритм виявлення цих станів буде складатися з таких кроків:

1) шукаємо немаркований процес P_i , для якого i -й рядок матриці R поелементно не перевищує вектор A , тобто для всіх $1 \leq k \leq m$ виконується умова: $R_{ik} \leq A_k$;

2) якщо такий процес знайдений, то додаємо i -й рядок матриці C до вектора A , маркуємо процес і повертаємося до кроку 1;

3) якщо таких процесів не існує, то робота алгоритму закінчується.

Отже, на першому кроці цей алгоритм шукає процес, який може бути виконаний до кінця, тобто всі необхідні для нього ресурси доступні, виконує його і повертає зайняті таким процесом ресурси в загальний фонд ресурсів, а сам процес маркує як завершений. Якщо ж виявиться, що всі процеси можуть працювати, то жоден з них у цей момент незаблокований.

Для ілюстрації роботи описаного алгоритму розглянемо рисунок 2.11.

Вектор існуючих ресурсів					Вектор доступних ресурсів					Матриця поточного розподілу	Матриця запитів
$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$ <div>НМС Плотери Сканери CD-ROM</div>					$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$ <div>НМС Плотери Сканери CD-ROM</div>					$C = \begin{bmatrix} 0010 \\ 2001 \\ 0120 \end{bmatrix}$	$R = \begin{bmatrix} 2001 \\ 1010 \\ 2100 \end{bmatrix}$

Рис. 2.11. Приклад використання алгоритму визначення взаємних блокувань

Тут є три процеси та чотири класи ресурсів (наприклад, накопичувачі на магнітній стрічці: НМС, плотери, сканери і пристрій для читання компакт-дисків). Процес 1 використовує один сканер. Процес 2 займає два НМС і пристрій для читання компакт-дисків. Процес 3 займає плотер і два сканери. І кожен з цих процесів потребує додаткового пристрою, як показує матриця R.

Працюючи з описаним вище алгоритмом виявлення взаємних блокувань, шукаємо процес, для якого запит ресурсів може бути виконаний. Вимоги першого процесу не можна виконати, тому що в системі немає доступного пристрою для читання компакт-дисків. Запити другого процесу також не можна задовольнити, оскільки немає вільних сканерів. Але третій процес може отримати все необхідне, тому він працює, завершується і повертає всі свої ресурси. Отже, $A = (2 \ 2 \ 2 \ 0)$.

З цього моменту може почати виконуватися процес 2, який після завершення повертає свої ресурси в систему. Отримаємо: $A = (4 \ 2 \ 2 \ 1)$.

Тепер може працювати і процес, який залишився. Отже, у розглянутій системі не виникає взаємного блокування.

Далі змінимо варіант, що зображений на рис. 2.11. Нехай процесу 3, крім двох НМС і плотера, знадобиться також і пристрій для читання компакт-дисків. Тоді виникає ситуація, коли вже жоден із запитів не може бути виконаним. Тобто вся система опиниться заблокованою.

Для попередження взаємних блокувань необхідно визначати *безпечні* та *небезпечні стани* взаємодії процесів.

Стан безпечний, якщо він не перебуває в глухому куті і якщо існує такий порядок планування, при якому кожен процес може виконатися до завершення, навіть якщо всі процеси захочуть отримати максимальну кількість необхідних їм ресурсів.

Небезпечний стан сам по собі ще не є блокованим. Однак тільки в безпечному стані система може гарантувати, що всі процеси успішно завершать свою роботу, а в небезпечному стані такої гарантії дати не можна.

Нідерландським ученим Е. Дейкстрою був розроблений **«алгоритм банкіра»**, який є розширенням алгоритму виявлення взаємних блокувань. Він перевіряє, чи веде виконання кожного з наявних запитів до небезпечного стану, і якщо так, то такий запит відхиляється.

«Алгоритм банкіра» для одного виду ресурсу. Щоб зрозуміти, чи є стан безпечним, перевіряється, чи можна надати достатньо ресурсів для завершення роботи процесу. Якщо так, то ці запити вважаються «погашеними», після чого перевіряється наступний, найближчий до межі запиту процес. Якщо в результаті всі запити можуть бути виконані, то такий стан визнається безпечним, а початковий запит можна задовольнити (виконати).

На рисунку 2.12(a) представлено чотири процеси, кожен з яких отримав певну кількість одиниць ресурсу. Усього є 10 одиниць ресурсу. В останній колонці міститься значення максимального запиту ресурсу кожним процесом.

Нехай у деякий момент виникає ситуація, що показана на рисунку 2.12(б). Цей стан є безпечним тому, що залишилися дві одиниці вільного ресурсу, й ОС може затримати всі запити, крім запитів процесу С, дозволяючи процесу С завершитися і повернути всі чотири одиниці ресурсу.

Розглянемо тепер, що могло б статися, якби в ситуації на рисунку 2.12(б) був виконаний запит ще однієї одиниці ресурсу для процесу В. Потрапили б у стан, що показаний на рисунку 2.12(в), який є небезпечним. Тоді якщо б усі процеси запросили свою максимальну кількість одиниць ресурсу, то ОС не змогла б їх забезпечити, і ми потрапили б у глухий кут, тобто відбулося б взаємне блокування.

Процес	Отримав	Max
A	0	6
B	0	5
C	0	4
D	0	7

Вільно: 10

а)

Процес	Отримав	Max
A	1	6
B	1	5
C	2	4
D	4	7

Вільно: 2

б)

Процес	Отримав	Max
A	1	6
B	2	5
C	2	4
D	4	7

Вільно: 1

в)

Рис. 2.12. Три стани розподілу ресурсів:

а) початкове; б) безпечне; в) небезпечне

«Алгоритм банкіра» для декількох видів ресурсів. Розглянутий вище «алгоритм банкіра» можна узагальнити і для управління системою з декількома видами ресурсів. Для цього:

1. Шукаємо в матриці R рядок, що відповідає процесу, у якого невиконані потреби ресурсів менше або дорівнюють вектору A. Якщо такого рядка не існує, то система в результаті потрапить у глухий кут, оскільки жоден процес не може пропрацювати до успішного завершення.

2. Припускаємо, що процес, рядок якого обрали в пункті 1, запросив усі необхідні ресурси і закінчив роботу. Зазначаємо цей процес як «завершений» і додаємо всі його ресурси до вектора A.

3. Повторюємо кроки 1 і 2, доки або всі процеси будуть помічені як «завершені» і стан у цьому випадку вважатиметься безпечним, або відбудеться взаємне блокування і тоді стан буде вважатися небезпечним.

2.5. Класичні проблеми міжпроцесної взаємодії

Проблема виробника і споживача. «Проблема виробника і споживача», відома як «проблема обмеженого буфера», полягає в такому. Нехай два процеси спільно використовують буфер обмеженого розміру. Один з них («виробник») поміщає дані в цей буфер, а інший («споживач») зчитує їх звідти.

Умови синхронізації:

- «виробник» повинен чекати, якщо буфер повний;
- «споживач» має чекати, якщо буфер порожній;
- операції з буфером – це критичні секції, тобто працювати з буфером може тільки один процес.

Нижче наведено вирішення «проблеми виробника і споживача» з використанням семафорів:

```
# define N 100 / * кількість сегментів у буфері */  
typedef int semaphore;  
semaphore mutex = 1; / * контроль доступу в критичну секцію */  
semaphore empty = N;  
/ * Кількість порожніх сегментів буфера */  
semaphore full = 0;  
/ * Кількість повних сегментів буфера */  
void producer (void)  
{  
    int item;  
    while (TRUE) {  
        item = produce__item ();  
        / * Створити дані, що поміщаються в буфер */  
        wait (& empty);  
        / * Зменшити лічильник порожніх сегментів буфера */
```

```

wait (& mutex); /* Вхід у критичну область */
insert__item (item);
    /* Помістити в буфер новий елемент */
signal (& mutex);
    /* Вихід з критичної області */
signal (full);
    /* Збільшити лічильник повних сегментів буфера */
}
}
void consumer (void);
{
    int item;
while (TRUE) {
    wait (& full);
        /* Зменшити кількість повних сегментів буфера */
    wait (& mutex); /* Вхід у критичну область */
    item = remove__item ();
        /* Видалити елемент з буфера */
    signal (& mutex);
        /* Вихід з критичної області */
    signal (& empty);
        /* Збільшити лічильник порожніх сегментів буфера */
    consume_item (item); /* обробка елемента */
}
}

```

Тут семафор *mutex* використовується для реалізації взаємного виключення, тобто для виключення одночасного звернення до буфера двох процесів. Решта семафорів використані для синхронізації: семафори *full* і *empty* необхідні, щоб гарантувати, що «виробник» припиняє роботу, коли буфер повний, а «споживач» припиняє роботу, коли буфер порожній.

Можна узагальнити цю задачу і на випадок *m*-виробників і *n*-споживачів.

Задача про «читачів» і «письменників». Це одна із задач синхронізації, що описує доступ процесів до поділюваної бази даних. Процеси при цьому розділяються на дві категорії: «читачі» – ніколи не модифікують базу даних і «письменники» – можуть читати і модифікувати базу даних.

Правила синхронізації тут такі:

- «читачі» можуть працювати паралельно, якщо ніхто з «письменників» не модифікує базу даних;
- модифікувати базу може тільки один «письменник», коли ніхто не читає інформацію з бази даних;
- змінні стану може переглядати та змінювати тільки один процес.

Нижче наведена схема вирішення цієї задачі:

«Читач»	«Письменник»
Чекати, доки закінчить роботу «письменник» (якщо хтось пише)	Чекати, доки закінчать роботу всі «читачі» (якщо хтось читає)
Читати з бази даних	Модифікувати базу даних
За необхідністю «розбудити» «письменника», що очікує	За необхідністю «розбудити» «читачів», що очікують

Задача про «філософів, що обідають». Ця задача моделює використання загальних ресурсів декількома процесами. Сутність її така. П'ять філософів сидять за круглим столом. На столі – п'ять тарілок з макаронами і п'ять виделок по одній між тарілками. Передбачається, що треба їсти двома виделками відразу. Поведінка кожного з філософів така: думає – хоче їсти (намагається взяти дві виделки) – їсть.

Обмеження:

- кожен філософ повинен чекати, доки не звільняться дві виделки поруч з ним;
- модифікація змінних стану повинна виконуватися у критичній секції.

Задача про «сплячого перукаря». Ця задача моделює обслуговування потоку запитів за чергою. Клієнти приходять у перукарню і встають до черги. Перукар запрошує та обслуговує клієнтів по одному.

Обмеження:

- клієнт повинен чекати, якщо є черга, і/або перукар зайнятий обслуговуванням іншого клієнта;
- перукар «спить», якщо немає клієнтів;
- перший клієнт «будить» перукаря.

2.6. Потоки

Визначення процесу, яке було описане вище, базується на двох незалежних концепціях: на групуванні ресурсів і на виконанні програм. Якщо ж їх розділити, то з'явиться поняття *поток*. Ресурсами керувати простіше, об'єднавши їх у формі процесу, однак, з іншого боку, процес можна розглядати як потік виконуваних команд або просто потік.

Потік — це незалежно планований контекст виконання, що розділяє єдиний адресний простір з іншими потоками свого процесу.

Елементи, що спільно використовуються потоками, — адресний простір, глобальні змінні, відкриті файли, дочірні процеси, сигнали і їх обробники, інформація про використання ресурсів.

Елементи, що окремо використовуються потоками, — лічильник команд, реєстри, стек, стан. Лічильник команд при цьому відстежує порядок виконання дій, реєстри зберігають поточні значення змінних, стек містить протокол виконання процесу і тимчасові змінні.

Хоча потік виконується всередині процесу, концепції процесу і потоку різні. Концепція потоків, зокрема, додає до моделі процесу можливість одночасного виконання в одному і тому самому середовищі процесу декількох достатньою мірою незалежних команд.

Порівняння процесів і потоків

Можливість	Процес	Потік
<i>Час для створення</i>	Більший	Менший
<i>Час для перемикання</i>	Більший	Менший
<i>Механізм взаємодії</i>	Складний	Простий
<i>Колективні дані</i>	Немає	Є
<i>Захист</i>	Є	Немає
<i>Положення</i>	Той самий або на різних комп'ютерах	Той самий комп'ютер

Потоки володіють деякими властивостями процесів, тому їх іноді називають «спрощеними процесами» (lightweight processes).

Багатопоточність – виконання декількох потоків в одному процесі.

У разі запуску багатопотокового процесу в системі з одним процесором потоки працюють по черзі. Ілюзія ж паралельної роботи потоків створюється шляхом постійного перемикання системи між потоками. І лише в багатопроцесорній або багатоядерній системі може бути реалізовано реальне паралельне виконання команд різних потоків.

Різні потоки в одному процесі не настільки незалежні, як різні процеси. У всіх потоків існує один адресний простір, що означає спільне використання глобальних змінних. Оскільки будь-який потік має доступ до будь-якої адреси комірки пам'яті в адресному просторі процесу, то один потік може змінювати інформацію в стеку іншого потоку. Захисту від цього не існує, оскільки, по-перше, це неможливо, а по-друге, це й не потрібно, оскільки потоки зазвичай виконують спільну задачу.

Якщо програма паралельно виконує багато різних дій і деякі з них можуть час від часу блокуватися, то використання декількох потоків дає такі переваги:

- спрощується сам програмний додаток;
- полегшується спільне використання даних;
- підвищується продуктивність (за наявності операцій введення / виведення);

- знижується навантаження на підсистему керування пам'яттю;
- з'являється можливість для повноцінного використання багатопроцесорних систем.

2.6.1. Основні операції з потоками та способи реалізації потоків

Створення нового потоку відбувається за допомогою функцій *pthread_create* – в UNIX, *CreateThread* – у Windows. Основним параметром цих функцій є ім'я процедури, яку необхідно запустити в новому потоці. У Windows також передбачена функція *CreateRemoteThread*, що дозволяє створити потік в іншому процесі.

Завершити потік можна зсередини самого цього потоку функціями *pthread_exit* – в UNIX або *ExitThread* – у Windows. Після цього потік зникає і планувальник його не використовує. Завершити потік можна також з іншого потоку за допомогою функцій *pthread_cancel* – в UNIX або *TerminateThread* – у Windows.

Очікування потоком завершення іншого певного потоку виконується за допомогою функцій *pthread_join* – у UNIX або *WaitForSingleObject* і *WaitForMultipleObject* – у Windows.

Потік може добровільно надати свою чергу іншому потоку за допомогою функцій *pthread_yield* – у UNIX або *SwitchToThread* і *SuspendThread* – у Windows.

Існують такі способи реалізації потоків:

- у просторі користувача;
- у просторі ядра;
- змішана реалізація.

У випадку реалізації потоків у просторі користувача пакет підтримки потоків цілком розміщується в просторі користувача. Ядро ОС при цьому нічого не знає про потоки і керує звичайними однопоточними процесами. Кожен процес тут має власну таблицю потоків. Коли потік збирається виконати дію, яка може призвести до локального блокування, він викликає процедуру з пакета

підтримки потоків. Ця процедура зберігає стан потоку в таблиці потоків, потім шукає в таблиці потік, готовий до запуску, і завантажує в реєстри його стан.

Реалізація потоків у просторі користувача забезпечує такі переваги:

- 1) можна реалізувати цей механізм в ОС, що не підтримує потоки;
- 2) переключення потоків відбувається на порядок швидше, ніж у режимі ядра;
- 3) кожен процес може мати свій власний алгоритм планування;
- 4) вище масштабованість додатків.

Проблеми, що виникають під час реалізації потоків у просторі користувача:

- 1) блокуючий системний виклик в одному з потоків зупинить усі потоки процесу;
- 2) у процесі запуску одного потоку жоден інший потік не буде запущений, доки перший потік добровільно не звільнить процесор.

У випадку реалізації потоків у просторі ядра таблиця потоків розташовується в ядрі, яке також містить звичайну таблицю процесів. Усі запити, які можуть блокувати потік, тут реалізуються як системні виклики. Коли один потік блокується, ядро ОС запускає інший потік з цього самого процесу або потік з іншого процесу. Недоліком такого підходу є збільшення часу перемикання потоків, оскільки всі операції з потоками реалізуються через системні виклики, а не за допомогою бібліотечних функцій.

У випадку змішаної реалізації потоків ядро ОС «знає» тільки про потоки свого рівня й управляє ними. Деякі потоки ядра при цьому можуть містити по декілька потоків користувацького рівня, які управляються так само, як потоки в системі, що не підтримує багатопоточність. У цьому підході поєднуються переваги перших двох методів.

2.6.2. Організація процесів і потоків в Linux

В UNIX-системах для кожного компонента образу процесу виділяють окрему ділянку пам'яті. Значимо, що образ процесу UNIX-системах 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.

Керуючий блок процесу в Linux відображається структурою даних *task_struct*, основні поля якої містять таку інформацію:

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

Також *task_struct* має кілька полів спеціалізованого призначення, що необхідні для різних підсистем Linux, а саме:

- 1) відомості для обробки сигналів;
- 2) інформацію для планування процесів;
- 3) інформацію про файли і каталоги, пов'язані з процесом;
- 4) структури даних для підсистеми керування пам'яттю.

Структура процесу містить інформацію, необхідну впродовж усього часу існування процесу (зокрема, коли він вивантажений на диск). Це такі дані, як *pid*, ефективний *uid*, пріоритет, покажчик на структуру користувача. Структури процесу поєднуються в черзі процесів у системі. Структура користувача містить інформацію, необхідну тільки під час виконання процесу (стан процесора, *uid*, *gid*, поточний каталог, інформацію про відкриті файли).

Необхідно зазначити, що в операційній системі Linux новий процес створюється шляхом копіювання атрибутів поточного процесу.

Новий процес може бути клонований (cloned), і при цьому такі ресурси, як файли, обробники сигналів і віртуальна пам'ять, використовуються сумісно. І, як наслідок, якщо два процеси використовують одну і ту саму віртуальну пам'ять, то вони функціонують як потоки в рамках одного і того самого процесу. Але для потоків структури даних окремо не задаються, отже, в операційній системі Linux потоки і процеси не розрізняють.

На рисунку 2.13 представлено модель процесів і потоків Linux.

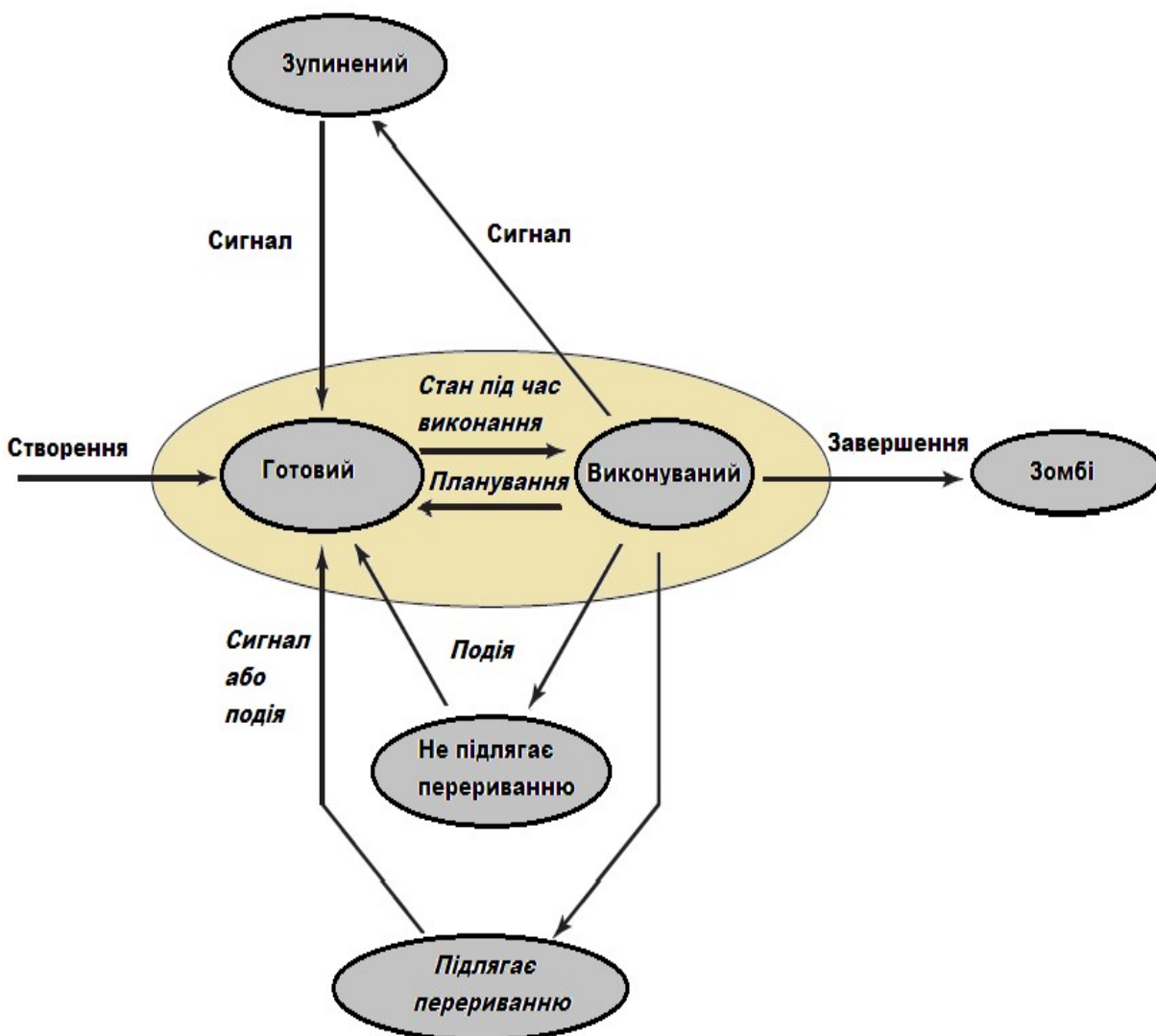


Рис. 2.13. Стани виконання процесу в Linux

Виконуваний (executing). Це стан, при якому поточний процес або виконується, або готовий до виконання.

Підлягає перериванню (interruptible). Це стан блокування, в якому процес очікує настання події, наприклад завершення операції введення / виведення, звільнення ресурсу або сигналу від іншого процесу.

Не підлягає перериванню (uninterruptible). Це стан блокування, при якому процес у неперервному стані безпосередньо очікує виконання певної апаратної умови, тому він не сприймає жодних сигналів.

Зупинений (stopped). Процес був зупинений і може бути продовжений тільки у разі відповідного впливу іншого процесу.

Зомбі (zombie). Процес був завершений, але з якоїсь причини його структура залишається в таблиці процесів.

2.7. Сигнали ОС UNIX

Сигнал в ОС UNIX – це асинхронне повідомлення процесу про будь-яку подію. Функції сигналів:

- оповіщення процесів про виникнення системних подій;
- забезпечення механізму для комунікації та синхронізації між прикладними процесами.

Сигнали дають можливість викликати спеціальну функцію – *обробник сигналу* під час виникнення певної події. Після обробки сигналу таким обробником основна програма продовжує роботу з того місця, в якому вона була перервана надходженням сигналу. Події позначаються цілими числами і представляються як символні константи.

Процес оповіщення про подію складається з двох етапів: *генерації сигналу і обробки сигналу*. У разі настання події, пов'язаної з будь-яким сигналом, ядро ОС генерує цей сигнал шляхом встановлення відповідного біта в масці сигналів, що

очікують у керуючому блоці процесу. При цьому процес може отримати сигнал лише в певні моменти часу:

- у разі вибору процесу диспетчером на виконання з черги готових;
- перед блокуванням або після блокування;
- під час деяких блокувань (наприклад, під час очікування введення з клавіатури).

Незалежно від джерела сигналу всі сигнали процесу доставляє ядро ОС.

Обробка сигналів. Кожен сигнал володіє дією, що встановлена за замовчуванням, яка виконується ядром, якщо процес не встановив для цього сигналу іншу дію.

Усього таких дій може бути п'ять:

- аварійне завершення зі створенням дампа пам'яті (*Dump*);
- аварійне завершення без створення дампа пам'яті (*Abort*);
- ігнорування сигналу (*Ignore*);
- призупинення процесу (*Stop*);
- відновлення роботи призупиненого процесу (*Continue*).

Процес може перевизначити дію за замовчуванням для будь-якого сигналу (крім сигналів *SIGKILL* і *SIGSTOP*) на ігнорування цього сигналу або на свій обробник сигналу. Процес також може в будь-який час вказати нову дію або скинути установки на дію за замовчуванням. Процес може тимчасово блокувати сигнал, і цей сигнал не буде оброблено, доки він не буде розблокований (функція *sigprogmask*).

Щоб вказати спосіб, яким необхідно обробити сигнал, варто виконати такі дії:

- 1) написати функцію обробника сигналу, включивши її в програму, для якої вона призначена;

2) зареєструвати цю функцію, викликавши на початку програми функцію *sigaction*. Виклики *sigaction* можна роити багаторазово, задаючи різні реакції на сигнал.

Нижче наведено приклад обробки сигналів під час натискання комбінації клавіш *Ctrl + C* (дією за замовчуванням для цього сигналу зазвичай є аварійне завершення програми):

```
include <signal.h>
.....
void handler (int sig);
{printf ("Отримано сигнал переривання \ n");
}
main ()
{   Struct sigaction act;
    act. sa__handler = handler;
    if (sigaction (SIGINT, & act, 0) != 0) {perror (...);
    exit (1); }
    /* Продовження роботи */
}
```

У таблиці 2.1 наведено 20 сигналів, що визначені в стандарті POSIX¹.

¹ Стандарт POSIX розроблено Інститутом інженерів з електротехніки й електроніки (IEEE), для того щоб надати можливість писати програми, що працюють у будь-якій UNIX-системі. Стандарт POSIX визначає мінімальний інтерфейс системних викликів, який має підтримувати всі системи, що сумісні з UNIX. Деякі ОС, що відрізняються від UNIX, тепер теж можуть підтримувати стандарт POSIX.

Сигнали, визначені в стандарті POSIX

№ з/п	Позначення сигналу	Дія за замовчуванням	Опис
1.	<i>SIGHUP</i>	<i>Abort</i>	Зависання термінала / аварійне зупинення керуючого процесу
2.	<i>SIGINT</i>	<i>Dump</i>	Переривання від клавіатури (Ctrl + C)
3.	<i>SIGQUIT</i>	<i>Dump</i>	Сигнал виходу з термінала (Ctrl + \)
4.	<i>SIGILL</i>	<i>Dump</i>	Неправильна команда
5.	<i>SIGTRAP</i>	<i>Dump</i>	Пастка трасування / точки зупину
6.	<i>SIGABRT</i>	<i>Dump</i>	Сигнал <i>Abort</i> від клавіатури
7.	<i>SIGBUS</i>	<i>Dump</i>	Помилка шини
8.	<i>SIGFPE</i>	<i>Dump</i>	Виключення з плаваючою точкою
9.	<i>SIGKILL</i>	<i>Abort</i>	Сигнал <i>Kill</i>
10.	<i>SIGUSR1</i>	<i>Abort</i>	Сигнал № 1, визначений користувачем
11.	<i>SIGSEGV</i>	<i>Dump</i>	Неправильна адреса пам'яті
12.	<i>SIGUSR2</i>	<i>Abort</i>	Сигнал № 2, визначений користувачем
13.	<i>SIGPIPE</i>	<i>Abort</i>	Запис у канал без його читання
14.	<i>SIGALRM</i>	<i>Abort</i>	Сигнал таймера від функції <i>alarm</i>
15.	<i>SIGTERM</i>	<i>Abort</i>	Сигнал завершення
16.	<i>SIGSTKFLT</i>	<i>Abort</i>	Помилка стека сопроцесора
17.	<i>SIGCHLD</i>	<i>Ignore</i>	Нащадок зупинений або завершений
18.	<i>SIGCONT</i>	<i>Continue</i>	Продовжено зупинений процес
19.	<i>SIGSTOP</i>	<i>Stop</i>	Процес зупинено
20.	<i>SIGTSTP</i>	<i>Stop</i>	Сигнал зупинки термінала (Ctrl + Z)

Під час обробки сигналу інші сигнали того самого типу блокуються. Тому обробники сигналів можуть не бути реєнтерабельними¹, оскільки декілька копій обробника не можуть

¹ Ядро Linux є реєнтерабельним. 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.

виконуватися одночасно. Однак обробник сигналу одного типу може перервати роботу оброблювача сигналу іншого типу.

Звичайні сигнали ядро не запам'ятовує в черзі. Якщо якийсь сигнал уже обробляється, то інший сигнал того самого типу, що надійшов у цей момент, просто губиться. У системах реального часу втрата сигналів є критичною. Тому багато версій UNIX (включаючи Linux) підтримують сигнали реального часу. Такі сигнали ядро поміщає в чергу, і вони не губляться – гарантується їх доставка в порядку FIFO.

Посилка сигналу процесу. Послати сигнал процесу можна за допомогою системного виклику `kill`:

```
# include <sys/types.h>
# include <signal.h>
int kill (pid_t pid, intsig);
```

Тут `pid` – це одержувач сигналу, а `sig` – номер сигналу. Якщо `pid > 0`, то сигнал `sig` надсилається процесові з $PID = pid$.

Якщо $pid = 0$, то сигнал посилається всім процесам у групі, до якої належить поточний процес. Якщо $pid = -1$, то сигнал посилається всім процесам у системі, крім першого (`init`), починаючи від найбільших PID у таблиці процесів. Якщо ж $pid < -1$, то сигнал посилається всім процесам у групі процесів `pid`.

Контрольні питання

1. Що таке процес?
2. Що таке мультипрограмування?
3. Назвіть причини для створення процесу в обчислювальних системах.
4. Які ситуації призводять до завершення процесу?
5. Які можливі переходи між станами процесів у моделі процесу з п'ятьма станами?
6. З яких елементів складається образ процесу?
7. Що входить до управляючого блоку процесу?
8. Що включають атрибути процесу?

9. Які дії виконує ОС під час створення процесу?
10. Опишіть ОС на основі ядра у складі користувацьких процесів.
11. Що таке планування процесів? Які є типи планування процесів?
12. У чому відмінність між цілісним і розподіленим планувальником? Які основні функції планувальника процесів?
13. Що таке безпріоритетна організація процесів? Які недоліки такої організації процесів?
14. За яким принципом відбувається робота дисципліни диспетчеризації FIFO?
15. За яким принципом відбувається робота дисципліни диспетчеризації LIFO?
16. За яким принципом відбувається робота дисципліни диспетчеризації FCFS?
17. За яким принципом відбувається робота дисципліни диспетчеризації SJN?
18. За яким принципом відбувається робота дисципліни диспетчеризації SRT?
19. За яким принципом відбувається робота дисципліни диспетчеризації RR?
20. Які переваги багатозадачності з витісненням на відміну від багатозадачності без витіснення?
21. Як може бути досягнуто гарантоване обслуговування процесів?
22. Які є види взаємодії процесів?
23. Які можливі проблеми під час взаємодії процесів?
24. Який механізм гарантовано запобігає виникненню взаємних блокувань у разі взаємодії процесів?
25. Які апаратні способи досягнення взаємного виключення?
26. Опишіть алгоритм Деккера. Який його недолік?
27. Для чого застосовуються примітиви синхронізації? Які існують примітиви синхронізації?
28. Що таке семафор? У якому випадку семафор називається сильним?

29. Наведіть приклад взаємного блокування з повторно використовуваними ресурсами.
30. Дайте визначення безпечних та небезпечних станів взаємодії процесів.
31. Назвіть і опишіть класичні проблеми міжпроцесної взаємодії.
32. Що таке потік? Якими елементами характеризується потік? Які переваги під час використання декількох потоків?
33. Які існують способи реалізації потоків?
34. Які є стани виконання процесів в Linux?
35. Що таке сигнал в ОС UNIX? Обробка сигналів UNIX.

РОЗДІЛ 3

УПРАВЛІННЯ ПАМ'ЯТТЮ В ОПЕРАЦІЙНИХ СИСТЕМАХ¹

Пам'ять в операційних системах – це ресурс комп'ютера, призначений для зберігання програмного коду і даних. Пам'ять зображають як масив машинних слів або байтів з їх адресами.

Існують різні види пам'яті, які перебувають у певній ієрархії. **Жорсткий диск комп'ютера** (допоміжний запам'ятовувальний пристрій – *secondary storage*) є найдешевшим і найповільнішим запам'ятовувальним пристроєм. **Оперативна пам'ять** зберігається в мікросхемах пам'яті, її називають **основною пам'яттю** (*main memory*), вона є більш швидкою та коштовною. Швидшими засобами зберігання даних є **кеші** процесора, обсяг цих кешів обмежений.

Керування пам'яттю в ОС – це досить складне завдання. Переважно потрібної за характеристиками пам'яті виявляється недостатньо, і щоб це не заважало роботі користувача, стає необхідним координувати між собою різні види пам'яті.

Управління пам'яттю поряд з управлінням процесами і ресурсами – одна з найбільш важливих функцій операційної системи.

Будь-яка програма, введена в систему, повинна бути розміщена в пам'яті й оформлена у вигляді процесу для її виконання. Кожна програма у разі введення в систему поміщається у вхідну чергу – сукупність процесів на диску, які очікують розміщення в пам'яті для виконання своїх програм. До свого виконання користувачькі програми проходять у системі декілька стадій.

Багатоетапна обробка програми, якою користуються

Вихідний код програми (у формі текстового файлу) мовою високого рівня або на асемблері перетворюється компілятором або асемблером в об'єктний модуль, що містить бінарні виконувані

¹ Під час підготовки цього розділу були використані літературні джерела [1–3], [4], [6], [15], [17], [21–23] та інші із списку використаних джерел.

машинні команди і таблицю символів, визначених і використаних у цьому модулі коду. Розглянута фаза називається часом компіляції.

Однак об'єктний модуль не може безпосередньо виконуватися, оскільки він містить недозволені посилання на зовнішні модулі та їх компоненти. Наступна фаза обробки програми – редагування зв'язків. Редактор зв'язків (linker) – системна програма, яка отримує на вхід один або декілька об'єктних модулів, а на виході видає завантажувальний модуль – двійковий код, утворений кодом декількох об'єктних модулів, в якому дозволені всі міжмодульні посилання, для кожного символу, зовнішнього для цього об'єктного модуля.

Завантажувальний модуль може бути завантажений у пам'ять для виконання за допомогою ще однієї системної програми – завантажувача (loader), який отримує на вхід завантажувальний модуль і файли з бінарними кодами системних бібліотек, які використовує програма. Завантажувач, об'єднуючи код програми з кодами системних бібліотек, створює бінарний образ програми в пам'яті.

Логічний та фізичний адресний простір

Концепція логічного адресного простору, пов'язаного з відповідним фізичним простором, є однією з основних для управління пам'яттю.

Логічною адресою називається адреса, що генерована процесором під час виконання машинної команди.

Фізична адреса – це реальна адреса в пам'яті, яку «бачить» і «розуміє» пристрій управління пам'яттю (Memory Management Unit – MMU).

Логічні адреси збігаються з фізичними у разі зв'язування адрес під час компіляції або завантаження (тобто до виконання програми). Однак у процесі з'єднання адрес під час виконання логічні адреси відрізняються від фізичних.

Пристрій управління пам'яттю

Пристрій управління пам'яттю (*Memory Management Unit – MMU*) – це один з модулів апаратури, що відповідає за адресацію пам'яті і пов'язаний з процесором та іншими пристроями системною шиною. З точки зору підтримки описаних концепцій адресації пристрій управління пам'яттю – це апаратура, перетворююча логічну адресу (отримана за загальною шиною від процесора) у фізичну (реальна адреса в пам'яті, за якою і відбувається звернення).

Апаратура MMU використовує значення регістра переміщення, що містить адресу початку області пам'яті, виділеної ОС для програми користувача. MMU додає значення регістра переміщення до (логічної) адреси, що згенерована програмою користувача, отримуючи в результаті фізичну адресу.

Програма користувача працює тільки з логічними адресами і не «бачить» фізичних адрес. Схема адресації і перетворення логічної адреси у фізичну з використанням регістра переміщення зображена на рисунку 3.1.

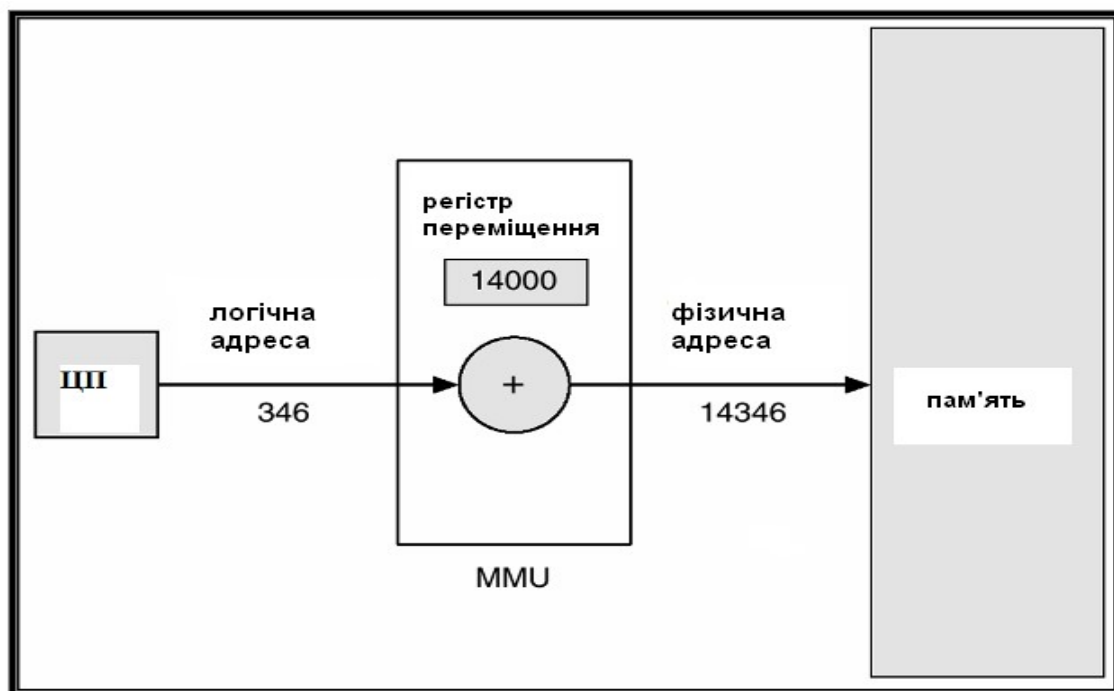


Рис. 3.1. Адресація з використанням регістра переміщення

Відкачування і підкачка

Користувацький процес може перебувати в різних станах під час обробки системою. Зокрема, процес може бути деякий час неактивним, якщо, наприклад, він виконується в режимі поділу часу, і користувач за терміналом обмірковує наступну команду або редагує вихідний код своєї програми. У подібних випадках процес може бути відкачано операційною системою на диск тому, що пам'ять, яку він займає, необхідна в цей момент для іншого активного процесу.

Відкачування і підкачка (*swapping*) – це дії операційної системи з відкачування (запису) образу неактивного процесу на диск або підкачування (зчитування) активного процесу в основну пам'ять. Потреба у виконанні подібних дій викликана нестачею основної пам'яті.

Файл відкачування (*backing store*) – область дискової пам'яті, використовувана операційною системою для зберігання образів відкачаних процесів. Файл відкачування організується максимально ефективно: забезпечується прямий доступ до всіх образів процесів у пам'яті (наприклад, через таблицю за номером процесу).

Популярний різновид стратегії відкачування і підкачки – ***roll out / roll in***: відкачування і підкачка на базі пріоритетів; більш пріоритетні процеси виконуються, менш пріоритетні – відкачуються на диск.

Під час відкачування найбільше часу витрачається у разі передачі даних: повний образ процесу може займати велику область пам'яті. Загальний час відкачування пропорційний розміру відкачуваних даних.

У поширених ОС: UNIX, Linux, Windows та ін., – реалізовані різні стратегії відкачування і підкачки.

Схема відкачування та підкачування зображена на рисунку 3.2.

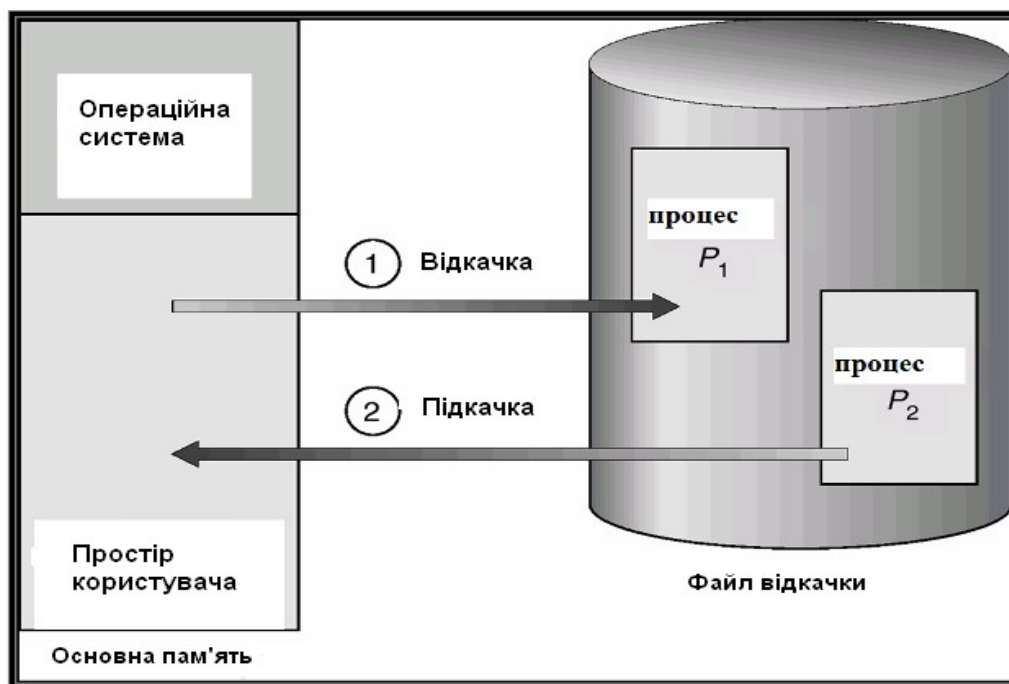


Рис. 3.2. Схема підкачки і відкачки

3.1. Технології розподілу пам'яті

Оперативна пам'ять (ОП) – це важливий ресурс ЕОМ. У ранніх ОС управління пам'яттю зводилося до завантаження програми і її даних із зовнішньої пам'яті в ОП.

Функції мультипрограмної ОС щодо управління пам'яттю:

- відстеження вільної і зайнятої пам'яті;
- виділення пам'яті процесам і звільнення пам'яті після завершення цих процесів;
- захист пам'яті;
- налаштування адрес програм;
- виділення динамічної пам'яті;
- переміщення програм між ОП і зовнішньою пам'яттю. Під час виконання останньої функції існує дві стратегії:

1) *свопінг*, або проста підкачка, – програми цілком завантажуються в ОП і цілком вивантажуються на диск;

2) програми частково завантажуються в ОП і частково вивантажуються на диск; якщо це прозоро для програміста, то така стратегія називається *віртуальною пам'яттю*, а якщо не прозоро, то – *оверлеєм*.

Основна операція щодо управління пам'яттю – це розміщення програми в ОП для її виконання процесором. У сучасних ОС при цьому використовується складна система, що називається *віртуальною пам'яттю*, яка використовує одну чи обидві базові технології: сегменти і сторінки. Але спочатку ознайомимося з більш простими технологіями.

3.1.1. Фіксований розподіл пам'яті

У цьому випадку пам'ять поділяється на області з фіксованими межами.

Розділи однакового розміру. Будь-який процес, розміри якого не перевищують розміру розділу, може бути завантажений у будь-який доступний розділ. Якщо ж усі розділи зайняті і немає жодного процесу в стані готовності або роботи, то ОС може вивантажити процес з будь-якого розділу і завантажити в нього інший процес, забезпечуючи процесор роботою. Прийняття рішення, який процес вивантажити, – це завдання планувальника.

Недолік цього способу – неефективне використання ОП, оскільки маленька програма цілком займає великий розділ. Це явище називається *внутрішньою фрагментацією*.

Розділи різного розміру. Цей спосіб не усуває повністю недолік, зазначений вище, але зменшує його вплив.

У разі призначення розділу процесу існує два підходи:

1) кожен процес можна розмістити в найменшому розділі, здатному повністю вмістити цей процес; це мінімізує внутрішню фрагментацію, але для кожного розділу потрібна своя черга планувальника;

2) кожен процес можна розмістити в найменшому доступному розділі, здатному повністю вмістити цей процес; при цьому використовується єдина черга планувальника.

Недоліки методу фіксованого розподілу:

- кількість розділів, що визначена в момент генерації системи, обмежує кількість активних процесів;

- оскільки розміри розділів встановлюються заздалегідь, це призводить до неефективного використання пам'яті.

Фіксований розподіл пам'яті є неефективним і тому не застосовується в сучасних ОС.

3.1.2. Динамічний розподіл

У цьому випадку в пам'яті формується змінна кількість розділів змінної довжини, а під час розміщення процесу в ОП для нього виділяється суто необхідна кількість ОП. Цьому методу властива *зовнішня фрагментація*, тобто фрагментованою стає пам'ять, зовнішня щодо всіх розділів. Для позбавлення від зовнішньої фрагментації ОС здійснює ущільнення: час від часу процеси переміщуються в пам'яті так, щоб вони займали суміжні області ОП. Для виконання такого ущільнення потрібен додатковий час і здатність програм до динамічного переміщення.

Під час організації динамічного розподілу використовується три основних алгоритми розміщення процесу в пам'яті:

1) *найкращий підходящий* – вибирається блок, розмір якого найбільш близький до необхідного;

2) *перший підходящий* – перевіряються всі вільні блоки з початку ОП і вибирається перший з них, достатній за розміром;

3) *наступний підходящий* – перевіряються всі вільні блоки, починаючи з того місця, де востаннє був виділений блок, і вибирається перший блок, достатній за розміром.

3.1.3. Система двійників

Як було зазначено, фіксований розподіл обмежує кількість активних процесів і неефективно використовує ОП за невідповідністю розмірів розділів і процесів. Динамічний розподіл вимагає більш складної організації і великих накладних витрат на ущільнення пам'яті.

Система двійників є компромісом між цими двома способами. У разі її використання ОП розподіляється блоками розміром $2k$, $L \leq k \leq U$, де $2L$ – мінімальний розмір блоку, що виділяється, а

$2U$ – максимальний блок, що розподіляється (розмір усієї пам'яті, що доступний для розподілу).

Спочатку весь доступний для розподілу простір ОП розглядається як єдиний блок розміру $2U$. Під час запиту блоку довільного розміру s , якщо $2U-1 \leq s \leq 2U$, то виділяється весь блок. В іншому випадку цей блок розділяється на два блоки-двійники з розмірами $2U-1$. Якщо тепер $2U-2 \leq s \leq 2U-1$, то за запитом виділяється один з двох цих двійників, в іншому випадку один з двійників знову ділиться навпіл. Цей процес продовжується доки, доки не буде згенеровано найменший блок, розмір якого не менше s .

При цьому система постійно веде список доступних блоків для кожного розміру $2i$. Доступний блок може бути вилучений зі списку $(2i + 1)$ поділом його навпіл і внесенням двох нових доступних блоків розміром $2i$ у список i . Коли ж пара двійників у списку i опиняється звільненою, вони видаляються зі списку та об'єднуються в єдиний блок у списку $(i + 1)$.

Модифікована версія системи двійників використовується для розподілу пам'яті ядром ОС Linux.

3.1.4. Проста сторінкова організація

У разі використання цього методу ОП розділяється на ряд кадрів однакового розміру.

Кожен процес також розділяється на сторінки однакового розміру, що збігається з розміром кадрів.

Процес завантажується шляхом завантаження всіх його сторінок у доступні, але не обов'язково суміжні кадри. При цьому відсутня зовнішня фрагментація, але є невелика внутрішня фрагментація, що є складовою частиною останньої сторінки процесу.

На рисунку 3.3 показано використання сторінок і кадрів.

№ кадру	Основна пам'ять	№ кадру	Основна пам'ять	№ кадру	Основна пам'ять
0	A.0	0	A.0	0	A.0
1	A.1	1	A.1	1	A.1
2	A.2	2	A.2	2	A.2
3	A.3	3	A.3	3	A.3
4	B.0	4	-	4	Д.0
5	B.1	5	-	5	Д.1
6	B.2	6	-	6	Д.2
7	C.0	7	C.0	7	C.0
8	C.1	8	C.1	8	C.1
9	C.2	9	C.2	9	C.2
10	C.3	10	C.3	10	C.3
11		11		11	Д.3
12		12		12	Д.4
13		13		13	
14		14		14	
а) завантаження процесів А, В, С		б) вивантаження процесу В		с) завантаження процесу Д	

Рис. 3.3. Завантаження процесів А, В, С, Д

Коли необхідно завантажити процес А (складається із чотирьох сторінок), ОС знаходить вільні кадри і завантажує в них сторінки процесу А.

Далі завантажується процес В (три сторінки) і процес С (чотири сторінки) (рис. 3.3[а]).

За умовою, процес Д має п'ять сторінок, тобто для завантаження йому не вистачає кадрів. Тому процес В буде призупинено і вивантажено (рис. 3.3[б]). Процеси блокуються і завантажується процес Д (рис. 3.3[с]).

Як бачимо, для завантаження процесу у разі простої сторінкової організації пам'яті необов'язковим є суміжне розташування кадрів, в які розміщуються сторінки одного процесу.

Для кожного процесу ОС підтримує таблицю сторінок, яка вказує розташування кадрів кожної сторінки процесу (рис. 3.4). Також формується таблиця вільних кадрів.

Таблиця сторінок містить по одному запису для кожної сторінки процесу. Ця таблиця індексується номером сторінки, починаючи з нуля, а кожний запис таблиці містить номер кадру в ОП. Операційна система підтримує єдиний список вільних кадрів.

Розділ 3. Управління пам'яттю в операційних системах

№ стор.	№ кадру	№ стор.	№ кадру	№ стор.	№ кадру	№ стор.	№ кадру	13
0	0	0	—	0	7	0	4	14
1	1	1	—	1	8	1	5	
2	2	2	—	2	9	2	6	
3	3			3	10	3	11	
						4	12	

Табл. сторінок процесу А Табл. сторінок процесу В Табл. сторінок процесу С Табл. сторінок процесу Д Список вільних кадрів

Рис. 3.4. Таблиці сторінок процесів

У програмі логічна адреса дорівнює номеру сторінки і зміщення всередині сторінки.

Фізична ж адреса дорівнює номеру кадру і зміщення.

Перетворення логічних адрес у фізичні – це завдання апаратного забезпечення.

На рисунку 3.5 наведено приклад трансляції логічної адреси з використанням простої сторінкової організації пам'яті для 16-бітових адрес.



Рис. 3.5. Приклад трансляції логічних адрес у фізичну адресу у разі сторінкової організації пам'яті

Отже, проста сторінкова організація подібна фіксованому розподілу. Її відмінність полягає в малому розмірі розділів, які до того ж можуть бути несуміжними. Розмір сторінки повинен бути кратним ступеню числа 2.

3.1.5. Проста сегментна організація

Тут кожен процес поділяється на ряд сегментів і завантажується шляхом завантаження всіх його сегментів у динамічні (не обов'язково суміжні) розділи. Логічна адреса у цьому випадку дорівнює номеру сегмента і зміщення. При цьому, порівняно з динамічним розподілом, відсутня внутрішня фрагментація, краще використовується пам'ять і знижені накладні витрати.

Цей спосіб подібний на динамічний розподіл пам'яті. Якщо в ньому не використовуються оверлеї і віртуальна пам'ять, то для виконання програми всі її сегменти повинні бути завантажені в ОП. Відмінність від динамічного розподілу полягає в тому, що сегменти програми можуть займати декілька розділів, до того ж несуміжних.

Цей спосіб усуває внутрішню фрагментацію, але породжує зовнішню фрагментацію. Сегментація у разі його використання прозора для програміста і застосовується у процесі розміщення коду і даних у різних сегментах для організації декількох незалежних адресних просторів. Як і під час сторінкової організації, тут використовується таблиця сегментів для кожного процесу і ведеться список вільних блоків ОП. Кожен запис таблиці сегментів при цьому повинен містити стартову адресу сегмента в ОП і його довжину.

Приклад трансляції адреси з використанням простої сегментної організації пам'яті для 16-бітових адрес показаний на рисунку 3.6.

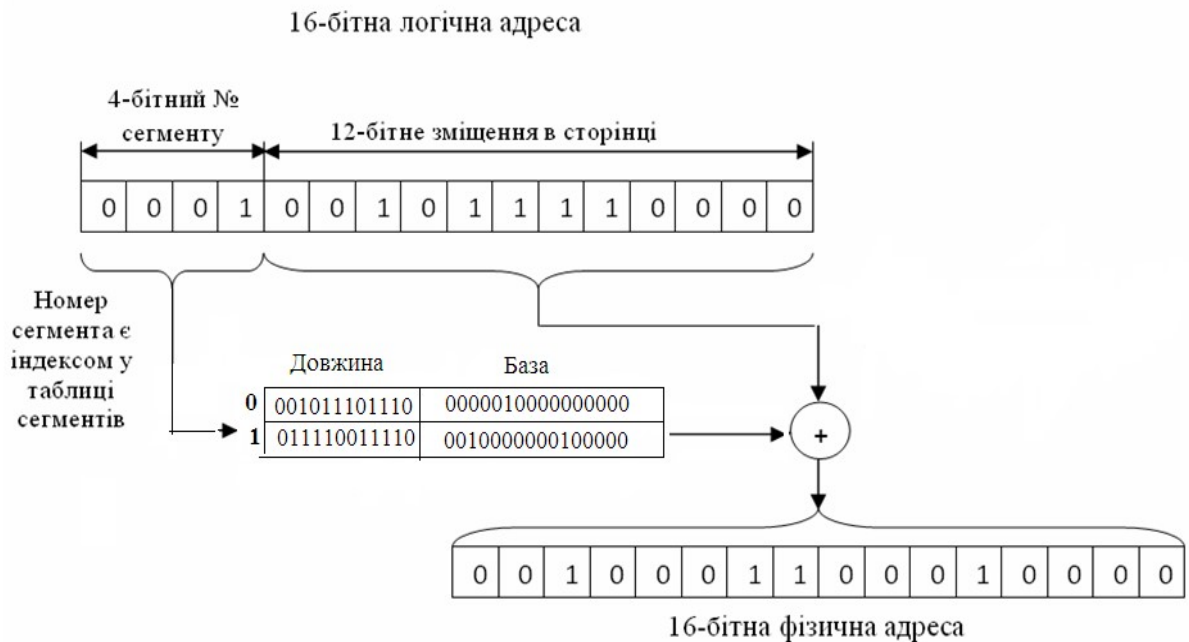


Рис. 3.6. Приклад трансляції фізичних адрес у логічну адресу у разі сегментації

3.2. Віртуальна пам'ять

3.2.1. Передумови для організації віртуальної пам'яті

Оскільки процес виконується тільки в основній пам'яті, ця пам'ять також називається *реальною*. Однак програміст або користувач мають справу з потенційно більшою за обсягом пам'яттю, виділеною на диску. Ця пам'ять називається **віртуальною**¹.

Передумови для організації віртуальної пам'яті:

1. Усі звернення до пам'яті в рамках процесу являють собою логічні адреси, які динамічно транслюються у фізичні адреси під час виконання. Це означає, що процес може бути вивантажений на диск і знову завантажений в ОП, так що під час роботи він може перебувати в різних місцях ОП.

¹ Вперше сповіщення про віртуальну пам'ять на основі сторінкової організації з'явилося 1962 року в роботі Kilburn I та ін. «*One-Level Storage System*», і незабаром після цього віртуальна пам'ять стала широко застосовуватись у комерційних системах.

2. Процес може бути розбитий на декілька частин (сторінок або сегментів), які не обов'язково повинні розташовуватися в ОП єдиним безперервним блоком. Це забезпечується завдяки використанню динамічної трансляції адрес, таблиць сторінок або сегментів.

Під час виконання цих умов одночасна наявність усіх сторінок або сегментів процесу в ОП не є необхідним.

Розглянемо, як це можна здійснити. Припустимо, що настав час завантаження нового процесу в пам'ять. ОС починає таке завантаження з розміщення в пам'яті тільки одного або декількох блоків (тут під блоком матимемо на увазі сторінку або сегмент), включаючи блок, що містить початок програми.

Частина процесу, що розташовується в деякий момент часу в ОП, називається *резидентною множиною процесу*. Далі під час виконання процесу за допомогою таблиці сторінок або сегментів процесор завжди може визначити, чи розташовується блок, до якого потрібно звернення, в ОП. Якщо ні, то процесор генерує переривання, що свідчить про відсутність блоку в пам'яті; тоді ОС переводить перерваний процес у заблокований стан і завантажує в ОП блок, що містить адресу, через яку відбулося переривання.

Після того, як необхідний блок завантажений в ОП, виконується переривання введення / виведення, що передає керування операційній системі, яка зі свого боку переводить заблокований процес у стан готовності, і процес продовжує виконання з перерваної команди.

Переваги використання віртуальної пам'яті:

- 1) в оперативній пам'яті може підтримуватися більша кількість процесів;
- 2) процес може бути більшим, ніж обсяг усієї фізичної пам'яті.

У таблиці 3.1 порівняно переваги і недоліки всіх видів організації пам'яті.

Таблиця 3.1

Порівняння різних технологій розподілу пам'яті

Технологія	Опис	Переваги	Недоліки
1	2	3	4
Фіксований розподіл	Основна пам'ять розділяється на ряд статичних розділів під час генерації системи. Процес може бути завантажений у розділ рівного або більшого розміру	Простота реалізації, малі системні накладні витрати	Неефективне використання пам'яті через внутрішню фрагментацію, фіксована максимальна кількість активних процесів
Динамічний розподіл	Розділи створюються динамічно, кожен процес завантажується в розділ суто необхідного розміру	Відсутня внутрішня фрагментація, більш ефективно використання основної пам'яті	Неефективне використання процесора через необхідність ущільнення для протидії зовнішній фрагментації
Проста сторінкова організація	Основна пам'ять розподілена на ряд кадрів однакового розміру. Кожен процес розподілений на деяку кількість сторінок однакового розміру тієї самої довжини, що і кадри. Процес завантажується шляхом завантаження всіх його сторінок у доступні, але не обов'язково послідовні, кадри	Відсутня зовнішня фрагментація	Є невелика внутрішня фрагментація
Проста сегментація	Кожний процес розподілено на ряд сегментів. Процес завантажується шляхом завантаження всіх своїх сегментів у динамічні (не обов'язково суміжні) розділи	Відсутня внутрішня фрагментація	Покращене використання пам'яті і знижені накладні витрати порівняно з динамічним розподіленням
Сторінкова організація віртуальної пам'яті	Усе, як під час простої сторінкової організації, з тим винятком, що не	Немає зовнішньої фрагментації; більш високий ступінь багатозадачності;	Накладні витрати через складність системи управління пам'яттю

1	2	3	4
	потрібно одночасно завантажувати всі сторінки процесу. Необхідні нерезидентні сторінки автоматично завантажуються в пам'ять	великий віртуальний адресний простір	
Сегментація віртуальної пам'яті	Усе, як під час простої сегментації, з тим винятком, що не потрібно одночасно завантажувати всі сегменти процесу. Необхідні нерезидентні сегменти автоматично завантажуються в пам'ять	Немає внутрішньої фрагментації; більш високий ступінь багатозадачності; великий віртуальний адресний простір; підтримка захисту і спільного використання	Накладні витрати через складність системи управління пам'яттю

Коли використовується віртуальна пам'ять, то віртуальні адреси не передаються напряму шиною пам'яті. Замість цього вони передаються диспетчеру пам'яті (*MMU – Memory Management Unit*), який відображує віртуальні адреси на фізичні адреси пам'яті (рис. 3.7). Тут диспетчер пам'яті показаний як частина мікросхеми процесора.

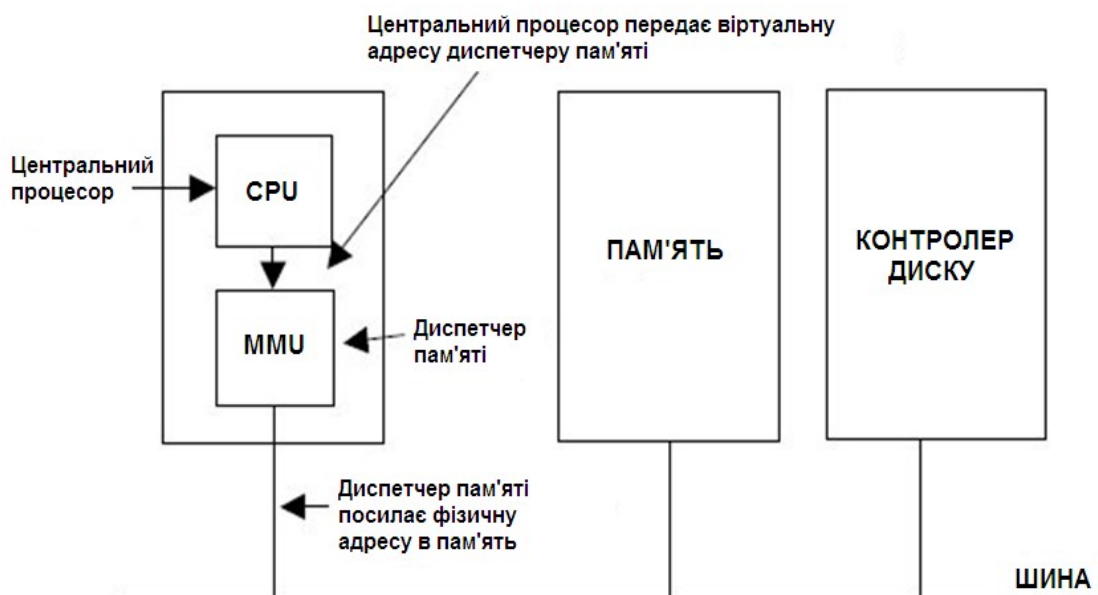


Рис. 3.7. Диспетчер пам'яті

Сьогодні реалізація віртуальної пам'яті може бути організована різними способами, що відрізняються в основному способом структуризації віртуального адресного простору. Виділяють три методи реалізації віртуальної пам'яті:

1. *Сторінкова віртуальна пам'ять* – організує переміщення даних між основною пам'яттю і диском сторінками – частинами віртуального адресного простору фіксованого і порівняно невеликого розміру.

2. *Сегментна віртуальна пам'ять* – передбачає переміщення даних сегментами – частинами віртуального адресного простору довільного розміру.

3. *Сегментно-сторінкова віртуальна пам'ять* – використовує дворівневий поділ, при якому віртуальний адресний простір поділяється на сегменти, а потім сегменти поділяються на сторінки. Одиницею переміщення даних є сторінка.

Для тимчасового зберігання сегментів і сторінок на диску відводиться спеціальна область або спеціальний файл (сторінковий файл або файл підкачки – paging file). Поточний розмір сторінкового файла є важливим параметром, що впливає на можливості ОС: чим більший сторінковий файл, тим більше додатків може одночасно виконувати ОС (щодо фіксованого розміру оперативної пам'яті). Але у разі збільшення кількості додатків, що одночасно працюють, збільшується і розмір сторінкового файла, а це сповільнює їх роботу, оскільки значна частина часу витрачається на переміщення даних на диск і у зворотному напрямку.

Розмір сторінкового файла в сучасних ОС є таким параметром, що налаштовується, і може бути визначений адміністратором системи для досягнення компромісу між рівнем програмування і швидкодією системи.

3.2.2. Сторінкова організація віртуальної пам'яті

Термін «віртуальна пам'ять» зазвичай асоціюється із системами, що використовують сторінкову організацію, хоча використовується і віртуальна пам'ять на основі сегментації.

Структура таблиці сторінок

Під час розгляду простої сторінкової організації вказувалося, що кожен процес має свою таблицю сторінок, що створюється операційною системою. Кожен запис у цій таблиці сторінок містить номер кадру відповідної сторінки в пам'яті. Таблиця сторінок розташовується в ОП.

Для організації віртуальної пам'яті кожний запис таблиці сторінок обов'язково має містити *біт присутності* P , що вказує на присутність відповідної сторінки в ОП; якщо ця сторінка розташовується в ОП, то в записі таблиці також міститься номер її кадру.

Іншим керуючим бітом у записі таблиці сторінок є *біт модифікації* M . Він вказує, чи було змінено вміст цієї сторінки з часу її останнього завантаження в ОП. Якщо таких змін не було, то у разі настання часу заміни сторінки в займаному нею у цей момент кадрі записувати цю сторінку на диск не потрібно, оскільки на диску вже є її точна копія.

Базовий механізм читання слова з пам'яті включає в себе трансляцію віртуальної, або логічної, адреси, що складається з номера сторінки і зміщення у фізичну адресу, яка являє собою номер кадру і зсув, з використанням таблиці сторінок.

У принципі кожна віртуальна адреса викликає звернення до двох фізичних адрес: одне звернення потрібно для вибірки відповідного запису з таблиці сторінок, а інше – для звернення до даних, що адресуються. Отже, проста схема віртуальної пам'яті, по суті, подвоює час звернення до ОП. Для подолання цієї проблеми більшість схем віртуальної пам'яті використовує спеціальний високошвидкісний кеш для записів таблиці сторінок, який зазвичай називається буфером швидкого перетворення адреси або просто буфером пошуку трансляції (TLB, Translation Lookaside Buffer). Цей кеш функціонує так само, як і звичайний кеш пам'яті.

3.2.3. Сегментація

Сегментація дозволяє програмісту розглядати пам'ять як область, що складається з безлічі адресних просторів або сегментів, які можуть мати різні (фактично динамічні) розміри. Звернення до пам'яті використовують адреси, що являють собою пари: номер сегмента, зміщення.

Така організація має ряд переваг порівняно з несегментованим адресним простором:

- 1) спрощується обробка зростаючих структур даних;
- 2) спрощується спільне використання коду і даних різними процесами;
- 3) покращується захист даних.

Як відомо, у разі простої сегментної організації кожен процес має власну таблицю сегментів, у кожному записі якої зазначена початкова адреса відповідного сегмента в ОП і його довжина. Та сама таблиця сегментів потрібна і щодо схеми віртуальної пам'яті, заснованої на сегментації, однак структура записів таблиці сегментів у цьому випадку ускладнюється.

Оскільки в ОП можуть перебувати не всі сегменти процесу, в кожному записі потрібна наявність *біта присутності*, який вказує на те, чи розташовується цей сегмент в ОП. Якщо сегмент розташований в ОП, то запис також включає його початкову адресу і довжину.

Ще один біт, необхідний у цій схемі, – *біт модифікації*, який вказує, чи було змінено вміст сегмента з часу його останнього завантаження в ОП. Якщо змін не було, то під час вивантаження сегмента немає потреби в його запису на диск.

Основний механізм читання слова з пам'яті включає в себе перетворення віртуальної, або логічної, адреси, що складається з номера сегмента і зміщення, у фізичну адресу з використанням таблиці сегментів, для зберігання якої використовується ОП. Коли запускається певний процес, в одному з регістрів зберігається стартова адреса його таблиці сегментів. Номер сегмента з віртуальною адресою використовується як індекс таблиці, що дозволяє

визначити початкову адресу сегмента. Для отримання ж фізичної адреси до початкової адреси сегмента додається зсув з віртуальної адреси.

3.2.4. Комбінація сегментації і сторінкової організації

І сторінкова організація, і сегментація мають свої переваги. Сторінкова організація, прозора для програміста, усуває зовнішню фрагментацію і таким способом забезпечує ефективне використання ОП. Оскільки переміщувані в ОП і з неї блоки мають фіксований, однаковий розмір, то полегшується створення ефективних алгоритмів управління пам'яттю. Сегментація ж є видимою для програміста і має свої переваги.

Деякі обчислювальні системи використовують переваги обох методів. У таких системах адресний простір користувача розбивається на ряд сегментів на розсуд програміста. Кожен сегмент зі свого боку розбивається на ряд сторінок фіксованого розміру, відповідного до розміру кадру ОП. Якщо розмір сегмента менше за розмір сторінки, то він займає сторінку цілком. З точки зору програміста, логічна адреса в цьому випадку складається з номера сегмента і зміщення в ньому. З позиції ж ОС зсув у сегменті варто розглядати як номер сторінки певного сегмента і зміщення в ній.

3.2.5. Алгоритми управління віртуальною пам'яттю

Алгоритми (стратегії) операційної системи, пов'язані з управлінням віртуальною пам'яттю, поділяються на такі групи.

Стратегія вибірки. Стратегія вибірки визначає, коли сторінка повинна бути передана в ОП. При цьому можливі два основні варіанти: *за вимогою і попередньо*. У разі вибірки *за вимогою* сторінка передається в ОП тільки тоді, коли виконується звернення до комірки пам'яті, розташованої на цій сторінці. У разі ж *попередньої вибірки* завантажуються не тільки сторінка, що викликала переривання звернення. Якщо сторінки процесу розташовані у вторинній пам'яті послідовно, то набагато більш

ефективним буде завантаження в ОП декількох послідовних сторінок за один раз, ніж завантаження цих самих сторінок по одній упродовж певного проміжку часу.

Стратегія розміщення. Стратегія розміщення визначає, де саме у фізичній пам'яті будуть розташовуватися частини процесу. У разі «чистої» сегментації стратегія розміщення використовує алгоритми першого відповідного, наступного, що підходить, та інші. Однак для систем, що використовують тільки сторінкову організацію або сторінкову організацію в поєднанні із сегментацією, стратегія розміщення зазвичай не досить важлива, оскільки апаратна трансляція адреси й апаратне звернення до пам'яті однаково результативні за будь-яких поєднань «сторінка-кадр».

Стратегія заміщення. Стратегія заміщення відповідає за вибір сторінок в ОП для їх заміщення сторінками, що завантажуються з вторинної пам'яті. Ця стратегія включає в себе ряд взаємопов'язаних питань: 1) яка кількість кадрів має бути виділена кожному активному процесу; 2) чи має безліч сторінок, які потенційно можуть бути заміщені сторінками, що завантажуються, обмежуватися одним процесом або як кандидати на заміщення можуть розглядатися всі кадри ОП; 3) які саме сторінки з розглянутої безлічі варто вибрати для заміщення.

Коли всі кадри ОП зайняті, а нам потрібно розмістити нову сторінку, стратегія заміщення визначає, яка із сторінок, що перебуває нині в ОП, повинна бути вивантажена, щоб звільнити кадр для сторінки, що завантажувється. Усі варіанти стратегії при цьому спрямовані на те, щоб вивантажити сторінку, звернень до якої в найближчому майбутньому не буде. Відповідно до принципу локалізації часто спостерігається сильна кореляція між сторінками, до яких останнім часом виконувалися звернення, і сторінками, до яких будуть звернення найближчим часом. Більшість стратегій намагається визначити майбутню поведінку програми на основі її минулої поведінки. Однак під час розгляду різних стратегій варто враховувати, що чим більш досконалий та інтелектуальний алгоритм використовує стратегія, тим вище будуть накладні витрати щодо його реалізації.

Оптимальний алгоритм полягає у виборі для заміщення тієї сторінки, звернення до якої виконуватиметься через найбільший проміжок часу порівняно з усіма іншими сторінками. Цей алгоритм призводить до мінімальної кількості переривань через відсутність сторінок. Зрозуміло, що реалізувати такий алгоритм на практиці неможливо, оскільки для цього системі потрібно знати всі майбутні події. Однак цей алгоритм є стандартом, з яким порівнюються реальні алгоритми.

Стратегія довше за всіх невикористовуваного елемента (*LRU, least Recently Used*) заміщає в пам'яті ту сторінку, звернень до якої не було довше, ніж до інших сторінок: згідно з принципом локалізації можна очікувати, що ця сторінка не буде використовуватися і в найближчому майбутньому. Ця стратегія близька до оптимальної. Основна проблема полягає у складності її реалізації.

Стратегія «першим увійшов – першим вийшов» розглядає кадри сторінок процесу як циклічний буфер з циклічним видаленням сторінок з нього. Це одна з найпростіших у реалізації стратегій заміщення, яка заміщає сторінку, що перебуває в ОП довше за інших. Однак не завжди ця сторінка використовується дійсно рідко.

Годинникова стратегія є компромісом між стратегією довше за всіх невикористовуваного елемента (близькою до оптимальної, але складною в реалізації) та стратегією «першим увійшов – першим вийшов» (простою в реалізації, але далекою від оптимальної). У найпростішій схемі годинникової стратегії з кожним кадром з'єднується один додатковий біт, що носить назву «біт використання». Коли сторінка вперше завантажується в кадр, її біт використання дорівнює 1. За наступних звернень до сторінки цей біт також дорівнює 1. Під час роботи алгоритму заміщення безліч кадрів розглядається як циклічний буфер, з яким пов'язаний покажчик, і у процесі заміщення сторінки покажчик переміщається до наступного кадру в буфері. Коли настає час заміщення сторінки, ОС сканує буфер для пошуку кадру, біт використання якого дорівнює 0, і щоразу, коли в процесі цього

пошуку зустрічається кадр з бітом використання, який дорівнює 1, він скидається в 0. Для заміщення вибирається перший, що зустрінеється кадр з нульовим бітом використання. Якщо ж усі кадри мають біт використання, який дорівнює 1, то показчик робить повний круг і повертається до початкового положення, замінюючи сторінку в цьому кадрі.

3.2.6. Управління пам'яттю в сучасних версіях Windows

Система керування віртуальною пам'яттю Windows контролює розподілення пам'яті і роботу сторінкової організації. Диспетчер пам'яті сконструйований для роботи на багатьох платформах і використовує сторінки розміром від 4 до 64 Кбайт. Платформи Intel і AMD 64 мають сторінки по 4 Кбайти, а платформа Intel Itanium має сторінки по 8 Кбайт.

Кожний користувацький процес Windows отримує окремий 32-бітовий адресний простір, що надає процесу до 4 Гбайт пам'яті. За замовчуванням частина пам'яті зарезервована для операційної системи так, що користувачу доступні 2 Гбайти віртуального адресного простору, і всі процеси сумісно використовують одні і ті самі 2 Гбайти системного простору.

Великі доданки, що інтенсивно використовують пам'ять (як клієнти, так і сервери), можуть запускатися більш ефективно в 64-бітному Windows. Крім нетбуків, найбільш сучасні персональні комп'ютери використовують архітектуру процесора AMD 64, який може запускатися і в 32-бітній і в 64-бітній системах.

Під час створення процесу в принципі йому повністю доступний простір користувача розміром 2 Гбайти (або 8 Тбайт у 64-бітному Windows). Цей простір розподіляється на сторінки фіксованого розміру, кожна з яких може бути завантажена в ОП.

Схема управління віртуальною пам'яттю вимагає як програмної, так і апаратної підтримки. Апаратна підтримка, що забезпечується процесором, включає динамічне перетворення віртуальних адрес у фізичні і генерацію переривання за відсутності сторінки, що адресується, або сегменту в ОП. Це переривання обробляється програмним забезпеченням управління пам'яттю.

Контрольні питання

1. Яку пам'ять називають основною пам'яттю (*main memory*)?
2. Що таке логічна і фізична адреси? Як відбувається перетворення логічної адреси у фізичну?
3. Що таке Memory Management Unit?
4. Які функції мультипрограмної ОС щодо управління пам'яттю?
5. Які є види фіксованого розподілу пам'яті? Чому такий розподіл пам'яті в чистому вигляді не використовується?
6. Коли виникає зовнішня фрагментація пам'яті? Як ОС зменшує негативний ефект зовнішньої фрагментації?
7. Що являє собою проста сторінкова організація основної пам'яті? Чому у разі такої організації пам'яті відсутня зовнішня фрагментація і зведена до мінімуму внутрішня?
8. Опишіть, як відбувається завантаження сторінок процесів у кадри пам'яті.
9. Що являє собою таблиця сторінок процесу? Навіщо створюються таблиці сторінок?
10. З чого складається логічна адреса у разі сторінкового розподілу пам'яті? Опишіть схему перетворення логічної адреси у фізичну.
11. З чого складається логічна адреса у разі сегментного розподілу пам'яті? Опишіть схему перетворення логічної адреси у фізичну.
12. Що таке віртуальна пам'ять? Коли вперше стало відомо про віртуальну пам'ять?
13. Які передумови виникнення технології віртуальної пам'яті? Які переваги і недоліки використання віртуальної пам'яті?
14. Які є методи реалізації віртуальної пам'яті? Якому з них надається найбільша перевага і чому?
15. Які керуючі біти мають бути в кожному записі таблиці сторінок під час віртуальної пам'яті?

16. Які є алгоритми (стратегії) управління віртуальною пам'яттю?
17. Опишіть алгоритм управління віртуальною пам'яттю «Стратегія вибірки».
18. Чому «Оптимальний алгоритм» реалізувати на практиці неможливо?
19. Яке розподілення пам'яті контролює система керування віртуальною пам'яттю Windows?
20. Який розмір простору користувача в 64-бітному Windows?

РОЗДІЛ 4

СИСТЕМА ВВЕДЕННЯ / ВИВЕДЕННЯ. ФАЙЛОВА СИСТЕМА¹

4.1. Система введення / виведення

Основні принципи програмного забезпечення введення / виведення

1. *Незалежність від пристроїв.* Цей принцип означає можливість написання програм, здатних отримувати доступ до будь-якого пристрою введення / виведення без попередньої вказівки конкретного пристрою.

2. *Однакове іменування.* Ім'я файлу або пристрою повинно являти собою просто текстовий рядок або ціле число і ніяк не повинно залежати від фізичного пристрою.

Способи здійснення введення / виведення

1. *Програмне введення / виведення.* Якщо, наприклад, потрібно роздрукувати на принтері рядок символів, то операційна система спочатку копіює ці дані в ядро. Потім ОС входить у цикл, на кожній ітерації якого друкує на принтері один символ. Після друку кожного символу процесор у циклі опитує готовність пристрою. Така поведінка процесора називається опитуванням, або активним очікуванням. Програмне введення / виведення легко реалізується, але його недолік полягає в тому, що центральний процесор зайнятий на весь час операціями введення / виведення.

2. *Введення / виведення, кероване перериваннями.* Коли виконується системний виклик друку рядка, буфер копіюється в простір ядра, а перший символ рядка копіюється на принтер. Після цього центральний процесор викликає планувальник, який запускає на виконання будь-який інший процес. Після того як принтер

¹ Під час підготовки цього розділу були використані літературні джерела [1–4], [9–11], [12], [15], [17], [21], [22] та ін.

надрукує символ і буде готовий прийняти наступний, принтер ініціює відповідне переривання. Воно перехоплюється обробником переривання і сигналізує ОС, що виведення символу завершено. Після цього ОС посилає на друк наступний символ тощо. Отже, під час друку символу на принтері процесор не зайнятий опитуванням готовності пристрою друку, а виконує іншу роботу.

3. Введення / виведення з використанням прямого доступу до пам'яті (DMA, Direct Memory Access). Очевидний недолік введення / виведення керованого перериваннями полягає в тому, що переривання відбуваються у разі друку кожного символу. Обробка цих переривань займає певний час, тому така схема не є ефективною. Рішення проблеми полягає у використанні DMA. Контролер DMA поставляє принтеру символи по одному, не займаючи при цьому центральний процесор. По суті, цей метод майже не відрізняється від програмного введення / виведення з тією лише різницею, що всю роботу замість центрального процесора виконує контролер DMA. Найбільший вигаш від використання DMA полягає у зменшенні кількості переривань з одного на кожний друкований символ до одного на весь друкований буфер.

4.1.1. Програмні рівні введення / виведення

Програмне забезпечення введення / виведення зазвичай організовано у вигляді чотирьох рівнів, показаних на рисунку 4.1, де у кожного рівня є свої функції і чітко визначений інтерфейс із сусідніми рівнями.

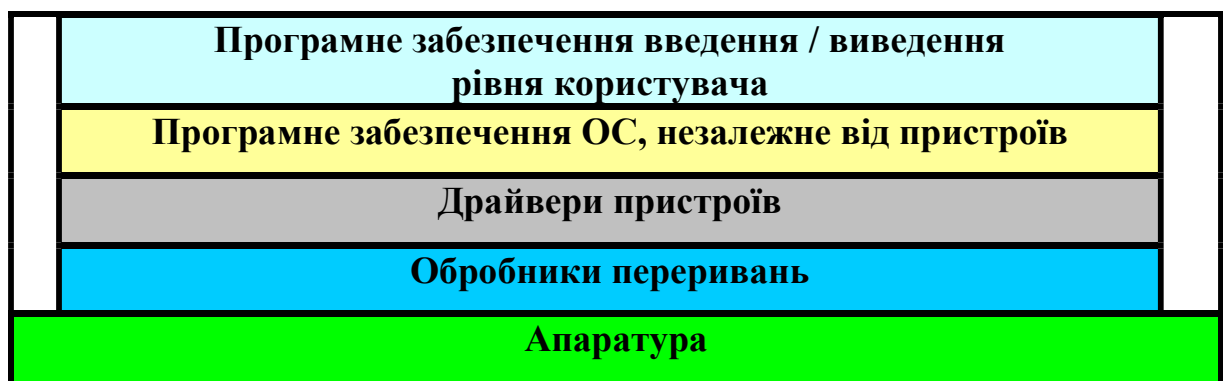


Рис. 4.1. Програмні рівні введення / виведення

Обробники переривань. Коли відбувається переривання, починає роботу обробник переривань. Після виконання необхідної роботи він може розблокувати драйвер, що запустив його.

Нижче наведені основні дії, що виконуються програмним забезпеченням після того, як відбулося апаратне переривання:

- 1) збереження всіх регістрів, що не збереглися апаратурою;
- 2) установка контексту для процедури обробки переривань (виконання цієї дії може включати встановлення TLB і таблиці сторінок);
- 3) установка покажчика стека для процедури обробки переривань;
- 4) видача підтвердження контролеру переривань (якщо ж централізованого контролера переривань немає, то дозвіл переривань);
- 5) копіювання вмісту регістрів у таблицю процесів;
- 6) запуск процедури обробки переривань, яка видобуває інформацію з регістрів контролера пристрою, що ініціював переривання;
- 7) вибір процесу, якому варто передати управління;
- 8) установка контексту блоку управління пам'яттю для наступного працюючого процесу;
- 9) завантаження регістрів для нового процесу;
- 10) запуск на виконання нового процесу.

Драйвери пристроїв. У кожного контролера є набір регістрів, які використовуються для того, щоб надавати команди певному пристрою і зчитувати стан цього пристрою. Команди, що видаються пристрою, залежать від конкретного типу цього пристрою, тому для управління кожним пристроєм введення / виведення потрібна спеціальна програма. Ця програма, називається драйвером пристрою і зазвичай пишеться виробником пристрою, причому для кожної операційної системи потрібні спеціальні драйвери. Кожен драйвер зазвичай підтримує тільки один тип пристроїв або ж клас близьких за типами пристроїв.

Щоб отримати доступ до реєстрів контролера, драйвер пристрою має бути частиною ядра ОС. Драйвери зазвичай розташовуються під всією іншою частиною операційної системи.

ОС зазвичай класифікує драйвери за декількома категоріями відповідно до типів пристроїв, що ними обслуговуються. До найбільш загальних категорій належать *блокові пристрої* (наприклад, диски, що містять блоки даних), до яких можлива незалежна адресація, і *символьні пристрої*, такі як клавіатура і принтери, які формують або приймають потік символів.

В ОС визначені стандартні інтерфейси, підтримувані всіма драйверами. Ці інтерфейси включають у себе набір процедур, які можуть викликатися з операційної системи для звернення до драйвера.

Драйвер пристрою повинен виконувати декілька функцій:

- обробку абстрактних запитів читання і запису, що надходять від незалежного від цього пристрою програмного забезпечення, розташованого над ним;
- перевірку вхідних параметрів;
- перетворення абстрактних параметрів у конкретні (наприклад, дисковий драйвер може перетворювати лінійний номер блоку в номери головки, доріжки і сектора);
- перевірку готовності і незайнятості пристрою;
- власне управління пристроєм – видачу йому необхідної серії команд;
- у багатьох випадках драйвер пристрою повинен чекати, доки контролер пристрою не виконає для нього певну роботу, тому він блокується, доки переривання від пристрою його не розблокує; в інших випадках операція завершується без затримок, і драйверу не потрібно блокуватися;
- після завершення виконання операції драйвер повинен перевірити, чи виконана ця операція без помилок. Якщо все в порядку, то драйвер передає дані незалежного від пристроїв програмного забезпечення.

Незалежне від пристроїв програмне забезпечення введення / виведення

Незалежне від пристроїв програмне забезпечення введення / виведення забезпечує такі функції:

- однаковий інтерфейс для драйверів пристроїв;
- буферизацію;
- повідомлення про помилки;
- захоплення і звільнення виділених пристроїв;
- розмір блоку, незалежний від пристрою.

Програмне забезпечення введення / виведення простору користувача

Програмне забезпечення введення / виведення простору користувача складається з бібліотек, приєднаних до програм користувача. Системні виклики введення зазвичай складаються з бібліотечних процедур. Наприклад, якщо програма мовою C містить виклик *write*, то бібліотечна процедура *write* буде скомпонована з програмою *i*, відповідно, міститиметься у виконуваному файлі.

Хоча більшість таких процедур в основному здійснюють звернення до системного виклику з відповідними параметрами, але є й ряд процедур введення / виведення, що виконують певну роботу. Зокрема, бібліотечними процедурами виконуються операції форматного введення і виведення (наприклад, функція *printf* мови C).

4.2. Завдання файлової системи. Поняття файла

4.2.1. Фізична організація файлової системи

Файл (file) – це поіменована область зовнішньої пам'яті, в яку можна записувати і з якої можна зчитувати дані (*суміжна область логічного адресного простору*). Основні цілі використання файла:

- довгострокове і надійне зберігання інформації;
- спільне використання інформації.

Кожен файл має свій тип, що визначає, яка інформація зберігається у файлі. Основні типи файлів: програма (код) або дані. Дані підрозділяються на числові, символні (текстові) і двійкові (довільна інформація).

Структура файла

У різних системах прийняті різні точки зору на структуру файлів. У ряді систем структура файла прив'язувалася до типу пристрою, на якому він міститься. У деяких інших системах структура файла була штучно ускладнена. Однак найбільш просту й уніфіковану точку зору з них запропонували автори системи UNIX: файл – це послідовність слів або байтів. Здавалося б, це очевидно, але перевага цього підходу до файлів у тому, що базове уявлення файла і базові операції над ним (read, write) не залежать від типу пристрою. Варто зазначити, що файли у своєму розвитку пройшли шлях, аналогічний розвитку архітектури комп'ютерів, – спочатку в бік значних ускладнень, потім – спрощення та уніфікації.

Файли можна умовно підрозділяти на файли *простої та складної структури* (хоча точка зору на структуру файла залежить від тієї програми, яка його обробляє).

Файли простої структури складаються з послідовності записів (*records*) – елементарних одиниць, у термінах яких виконуються операції обміну з файлом. Записи можуть бути: рядками, якщо це текстовий файл; двійковими даними фіксованої довжини; двійковими даними змінної довжини.

Файли складної структури можуть бути будь-якого виду, наприклад: відформатованим документом Microsoft Office (такий файл, крім, власне, тексту, містить керуючі символи перемикання шрифтів, кольорів тощо); завантажувальним модулем реального або віртуального двійкового коду, зокрема portable executable (PE)-файлом для платформи NET; class-файлом для платформи Java; подібні файли складаються з декількох секцій, містять внутрішні посилання і таблиці та ін.

Складна структура файла може бути змодельована записами шляхом додавання відповідних керуючих символів.

Файли інтерпретуються операційною системою або програмами їх обробки.

Файлова система (ФС) – це частина операційної системи, що включає в себе:

- сукупність усіх файлів на диску;
- набори структур даних, використовуваних для управління файлами;
- комплекс системних програмних засобів, що реалізують операції над файлами.

ФС відіграє роль «проміжного шару», що усуває для користувача / програміста всі складнощі фізичної організації довготривалого сховища даних і створює для програм більш просту логічну модель цього сховища, а також надає їм набір зручних у використанні команд для маніпулювання файлами.

Завдання, розв'язувані файловою системою, залежать від способу організації обчислювального процесу в цілому.

Основні функції ФС в однокористувацьких і однопрограмих ОС (типу MS-DOS):

- іменування файлів;
- програмний інтерфейс для додатків;
- відображення логічної моделі файлової системи на фізичну організацію сховища даних;
- стійкість файлової системи до збоїв харчування, помилок апаратних і програмних засобів.

В однокористувацьких мультипрограмих ОС також мають бути передбачені засоби для блокування файла і його частин, запобігання гонок, виключення глухих кутів, узгодження копій тощо.

У багатокористувацьких ОС з'являється ще одна задача – захист файлів одного користувача від несанкціонованого доступу іншого користувача.

Нарешті, ще більш складними стають функції ФС, яка працює у складі мережевої ОС.

Принципи розміщення файлів, каталогів і системної інформації на реальному пристрої описуються фізичною організацією файлової системи. Різні ФС мають будь-яку фізичну організацію.

4.2.2. Будова диска

Основним типом пристрою для зберігання файлів є дискові накопичувачі. Ці пристрої призначені для зчитування і запису даних на жорсткі і гнучкі магнітні диски.

Жорсткий диск складається з однієї або декількох скляних чи металевих пластин, кожна з яких покрита з одного або з двох боків магнітним матеріалом. На кожному боці пластини шляхом відповідного намагнічування поверхні магнітного шару розмічені тонкі концентричні кільця-доріжки (tracks), на яких зберігаються дані (рис. 4.2).

Коли диск обертається, магнітна головка накопичувача зчитує двійкові дані з обраної магнітної доріжки або записує їх на магнітну доріжку. Для цього головка може позиціонуватися над заданою доріжкою. Зазвичай усі головки закріплені на єдиному механізмі, що переміщується, і рухаються синхронно. Тому коли головка фіксується на заданій доріжці однієї поверхні, всі інші головки зупиняються над доріжками інших пластин з такими самими номерами. Сукупність доріжок одного і того самого радіуса на всіх поверхнях усіх пластин дискового пакета називається *циліндром*.

Кожна доріжка розбивається на фрагменти, названі секторами або блоками, так що всі доріжки мають рівну кількість секторів, в які можна максимально записати одну і ту саму кількість байтів. Сектор має фіксований для конкретної ОС розмір, що виражається ступенем числа 2; найчастіше розмір сектора становить 512 байтів. Враховуючи ж, що доріжки різного радіуса мають однакову кількість секторів, щільність запису стає тим більшою, чим ближче розташовується доріжка до центра диска.

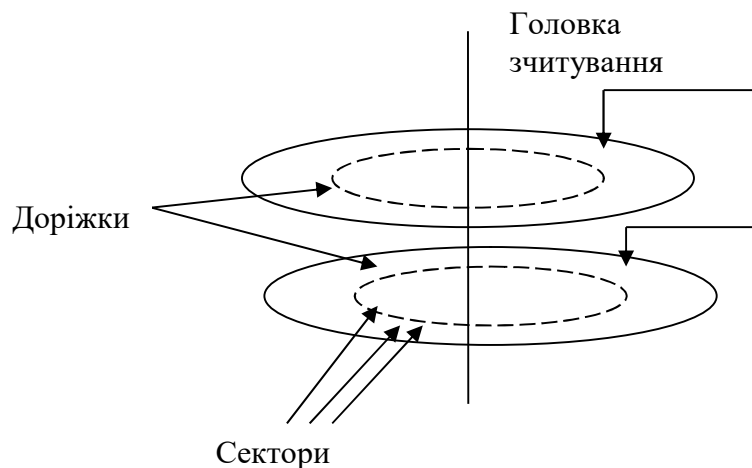


Рис. 4.2. Схема устрою накопичуваних жорстких магнітних дисків

Сектор – це найменша адресуєма одиниця обміну даними дискового пристрою з оперативною пам'яттю. Щоб контролер міг знайти на диску потрібний сектор, необхідно задати йому всі складові адреси цього сектора: номер циліндра, номер поверхні, номер доріжки та, власне, номер сектора на цій доріжці.

ОС під час роботи з диском використовує, як правило, власну одиницю дискового простору, звану *кластером* (в UNIX кластер називається блоком). У разі створення файлу місце на диску йому виділяється кластерами. Доріжки і сектори створюються в результаті виконання процедури фізичного, або *низькорівневого форматування*, що передує використанню диска. Для визначення меж блоків на диск записується ідентифікаційна інформація. Низькорівневий формат диска не залежить від типу ОС, яка цей диск використовуватиме. Розмітку диска під конкретний тип файлової системи виконують процедури високорівневого, або *логічного форматування*. Під час високорівневого форматування визначається розмір кластера, а на диск записується інформація, необхідна для роботи файлової системи, зокрема відомості про доступний і невикористаний простір, про межі областей, відведених під файли і каталоги, про пошкоджені області. Крім того, на диск записується *завантажувач операційної системи* – невелика програма, яка починає процес ініціалізації операційної системи після включення живлення або перезавантаження комп'ютера.

Перш ніж форматувати диск під певну файлову систему, його можна розбити на розділи. *Розділ* – це безперервна частина фізичного диска, яку операційна система являє користувачу як окремий логічний пристрій (часто використовуються також назви «логічний диск» і «логічний розділ»). Логічний пристрій функціонує так, нібито це окремий фізичний диск. ОС різного типу зазвичай використовують єдине для всіх них уявлення про розділи, але створюють на його основі логічні пристрої, специфічні для кожного типу ОС. Так само як файлова система, з якою працює одна ОС, у загальному випадку не може інтерпретуватися ОС іншого типу, одні й ті самі логічні пристрої часто не можуть бути використані операційними системами різного типу. При цьому на кожному логічному пристрої може створюватися лише одна файлова система, а на різних логічних пристроях одного і того самого фізичного диска можуть розташовуватися файлові системи різного типу.

Файлова система FAT

Абревіатура FAT (file allocation table) розшифровується як «таблиця розміщення файлів». Цей термін належить до лінійної табличної структури з відомостями про файли – іменами файлів, їх атрибутами й іншими даними, що визначають місцезнаходження файлів (чи їх фрагментів) у середовищі FAT. Елемент FAT *визначає* фактичну область диска, у якій зберігається початок фізичного файла.

У файловій системі FAT логічний дисковий простір будь-якого логічного диска поділяється на дві області (рис. 4.3): *системну область* і *область даних*.

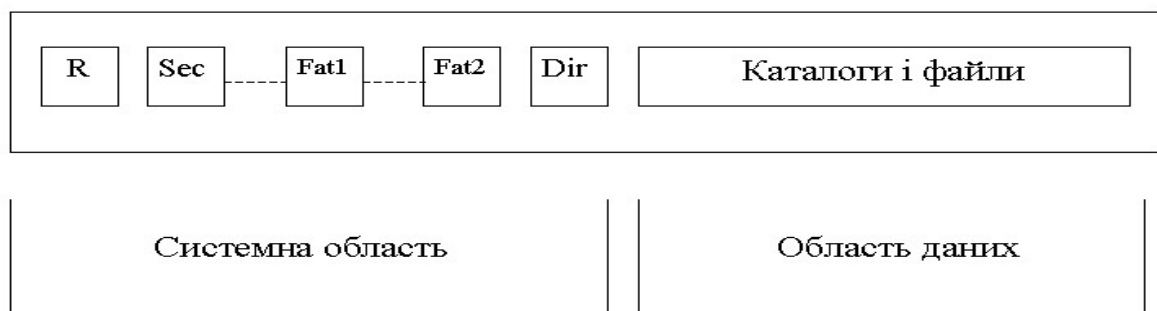


Рис. 4.3. Структура логічного диска

Системна область логічного диска створюється й ініціалізується під час форматування, а потім оновлюється у процесі маніпулювання файловою структурою. Область даних логічного диска містить файли і каталоги, підпорядковані кореневому. Вона, на відміну від системної області, доступна через користувацький інтерфейс DOS. Системна область складається з таких компонентів, розміщених у логічному адресному просторі підряд:

- завантажуючого запису (*boot record, BR*);
- зарезервованих секторів (*reserved sector, ResSecs*);
- таблиці розміщення файлів (*file allocation table, FAT*);
- кореневого каталогу (*root directory, Rdir*).

Також є файлові системи VFAT (віртуальна FAT) і FAT32, NTFS.

У назву файлової системи **NTFS** входять слова «*New Technology*», тобто «нова технологія». Дійсно, NTFS містить ряд значних удосконалень і змін, що істотно відрізняють її від інших файлових систем. З погляду користувачів, файли, як і раніше, зберігаються в каталогах (часто званих «папками», чи *фолдерами*, в середовищі Windows). У NTFS, на відміну від FAT, робота на дисках великого обсягу відбувається набагато ефективніше; є засоби для обмеження в доступі до файлів і каталогів, введені механізми, що істотно підвищують надійність файлової системи, знято багато обмежень на максимальну кількість дискових секторів і/чи кластерів.

Основні можливості файлової системи NTFS

Під час проектування системи NTFS особлива увага була приділена таким характеристикам:

- **Надійність.** Високопродуктивні комп'ютери і системи спільного користування (сервери) повинні мати підвищену надійність, що є ключовим елементом структури і поведінки NTFS. Одним зі способів збільшення надійності є введення механізму транзакцій, при якому здійснюється **журналювання** файлових операцій.

- *Розширена функціональність.* NTFS проєктувалася з урахуванням можливого розширення. У ній були втілені багато додаткових можливостей: удосконалена відмовостійкість; емуляція інших файлових систем; могутня модель безпеки; рівнобіжна обробка потоків даних і створення файлових атрибутів, обумовлених користувачем.

- *Підтримка POSIX.* Оскільки уряд США вимагав, щоб усі закупаювані ним системи хоча б у мінімальному ступені відповідали стандарту POSIX, така можливість була передбачена і в NTFS. До базових засобів файлової системи POSIX належить не обов'язкове використання імен файлів з урахуванням регістра, збереження часу останнього звертання до файла і механізм так званих «жорстких посилань» – альтернативних імен, що дозволяють посилатися на той самий файл двома і більше іменами.

- *Гнучкість.* Модель розподілу дискового простору в NTFS відрізняється надзвичайною гнучкістю. Розмір кластера може змінюватися від 512 байт до 64 Кбайт; він являє собою число, кратне внутрішньому кванту розподілу дискового простору. NTFS також підтримує довгі імена файлів, набір символів Unicode та альтернативні імена формату 8.3 для сумісності з FAT.

4.2.3. Фізична організація та адресація файла

Фізична організація файла – це спосіб розміщення файла на диску. Основними критеріями ефективності фізичної організації файлів є:

- швидкість доступу до даних;
- обсяг адресної інформації файла;
- ступінь фрагментації дискового простору;
- максимально можливий розмір файла.

Безперервне розміщення – це найпростіший варіант фізичної організації, при якому файлу надається послідовність кластерів диска, що утворюють безперервну ділянку дискової пам'яті. Цей метод має свої переваги: 1) висока швидкість доступу, оскільки витрати на пошук і зчитування кластерів файла мінімальні;

2) мінімальний обсяг адресної інформації; 3) необмежений максимально можливий розмір файла.

Однак реалізувати цю схему складно через те, що розмір файла, як правило, заздалегідь невідомий. Ще більш серйозною проблемою є фрагментація: через деякий час після створення такої файлової системи в результаті виконання численних операцій створення та видалення файлів простір диска неминуче перетворюється на «клаптикову ковдру», що включає в себе значну кількість вільних областей невеликого розміру.

Розміщення файла у вигляді зв'язаного списку кластерів, коли на початку кожного кластера міститься покажчик на наступний кластер.

На відміну від попереднього способу, кожен кластер тут може бути приєднаний до ланцюжку кластерів будь-якого файла, а отже, фрагментація на рівні кластерів відсутня. Файл може змінювати свій розмір під час існування, нарощуючи кількість кластерів. Але маючи певні переваги, цей метод також має свої недоліки: 1) складність реалізації доступу до довільного місця файла; 2) кількість даних файла, що містяться в одному кластері, не дорівнює ступеню числа 2 (оскільки одне машинне слово вже витрачено на номер наступного кластера), а багато програм читають дані кластерами, розмір яких дорівнює ступеню числа 2.

Використання зв'язаного списку індексів. Це – модифікація попереднього способу. Тут файлу також виділяється пам'ять у вигляді зв'язаного списку кластерів. При цьому номер першого кластера запам'ятовується в записі каталогу, де зберігаються характеристики цього файла, а інша адресна інформація відділена від кластерів файла. З кожним кластером диска зв'язується особливий елемент – індекс, де всі індекси розташовуються в окремій області диска: в ОС MS-DOS це таблиця FAT, що займає один окремий кластер. Коли пам'ять вільна, всі індекси мають нульове значення. Якщо ж деякий кластер N призначений деякому файлу, то індекс цього кластера дорівнює або номеру M наступного

кластера цього файла, або приймає спеціальне значення, яке є ознакою, що цей кластер є для файла останнім. Індекс попереднього кластера файла приймає значення N, вказуючи на новопризначений кластер.

За такої фізичної організації зберігаються всі переваги попереднього способу: мінімальність адресної інформації, відсутність фрагментації і відсутність проблем під час зміни розміру файла, але з'являються і додаткові переваги:

- висока швидкість доступу (для доступу до довільного кластера файла не потрібно послідовно зчитувати його кластери, достатньо прочитати тільки сектори диска, які містять таблицю індексів, відрахувати потрібну кількість кластерів файла за ланцюжком і визначити номер потрібного кластера);
- дані файла заповнюють кластер цілком, а отже, мають обсяг, рівний ступеню числа 2.

Недолік же цього методу тільки один: уся таблиця FAT повинна постійно перебувати в ОП або щоразу цілком завантажуватися в ОП.

Перелік номерів кластерів, зайнятих файлом (метод і-вузлів). Такий перелік і слугує адресою файла. З кожним файлом пов'язана структура даних «і-вузол», що містить також атрибути файла.

Недолік цього методу: довжина адреси залежить від розміру файла, і для великого файла може становити значну величину. Але і переваги цього методу в тому, що: 1) висока швидкість доступу до довільного кластера файла, оскільки тут застосовується пряма адресація, яка виключає перегляд ланцюжка покажчиків під час пошуку адреси довільного кластера файла; 2) відсутня фрагментація на рівні кластерів; 3) кожен вузол повинен міститися в ОП тільки тоді, коли відповідний йому файл відкритий.

Модифікований підхід до переліку номерів кластерів. Цей метод використовується в ОС UNIX у традиційних файлових системах s5 і ufs, в ОС Linux у найбільш поширених файлових

системах *ext2fs*¹ та *ext3fs* (остання відрізняється від першої тільки наявністю журналювання). Для скорочення обсягу адресної інформації прямий спосіб адресації поєднується з непрямым.

У стандартній для UNIX ФС *ufs* використовується наступна схема адресації кластерів файла (рис. 4.4). Для зберігання адреси файлу виділено 15 полів, кожне з яких складається з 4 байтів. Якщо розмір файла менше або дорівнює 12 кластерам, то номери цих кластерів безпосередньо перераховуються в перших дванадцяти полях адреси.

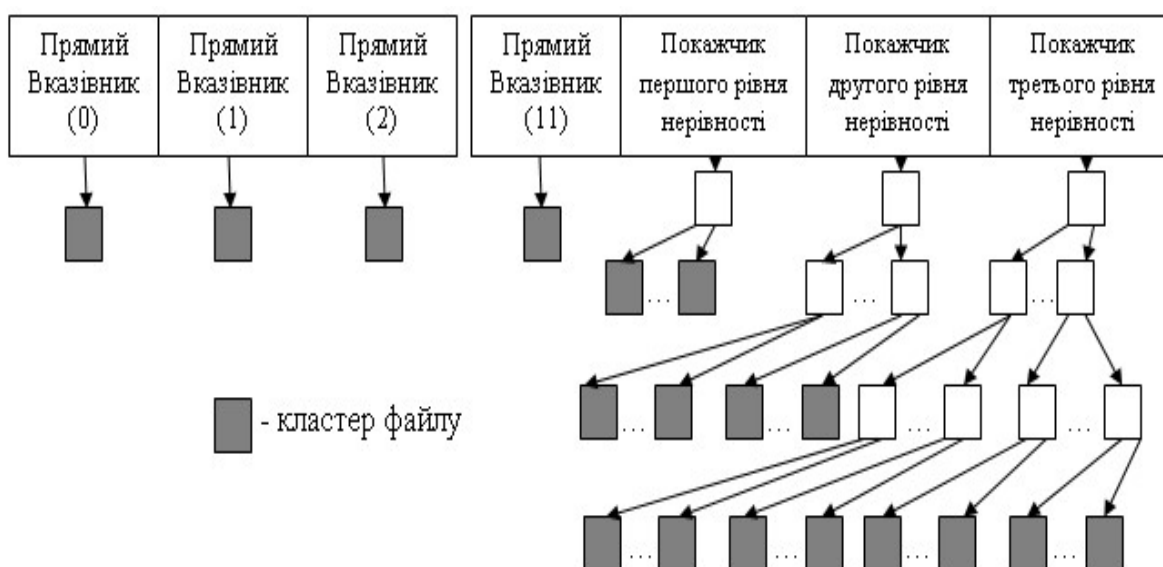


Рис. 4.4. Схема адресації кластерів файла у файловій системі *ufs*

Якщо розмір файла перевищує 12 кластерів, то наступне, 13-є поле містить адресу кластера, в якому можуть бути розташовані номери наступних кластерів файла. Якщо розмір файла перевищує $12 + 2\,048 = 2\,060$ кластерів, то використовується 14-є поле, в якому перебуває номер кластера, що містить 2 048 номерів кластерів, кожен з яких зберігає 2 048 номерів

¹ *ext2fs* – стандартна дискова файлова система для Linux, в якій дисковий простір ділять не на групи циліндрів, а просто на групи блоків (block groups), не прив'язані до геометрії диска. Такі групи за структурою подібні на групи циліндрів: кожна група блоків теж є зменшеною копією файлової системи із суперблоком, таблицею індексних дескрипторів тощо.

кластерів даних файла, тобто тут застосовується вже подвійна непряма адресація. І нарешті, якщо файл включає більше $12 + 2\,048 + 2\,048 \times 2\,048 = 4\,196\,364$ кластерів, то використовується останнє 15-е поле для потрійної непрямої адресації.

Метод переліку адрес кластерів файла задіяний і у файловій системі NTFS, яка використовується в ОС сімейства Microsoft Windows NT. Тут він доповнений досить природним прийомом, що скорочує обсяг адресної інформації: адресуються не кластери файла, а безперервні області, що складаються із суміжних кластерів диска. Кожна така область, що називається групою (в *ext2fs*) або екстентом (*extent*), описується за допомогою двох чисел: початкового номера кластера та кількості кластерів у відрізку. Оскільки для скорочення часу операції обміну ОС намагається розмістити файл у послідовних кластерах диска, то в більшості випадків кількість послідовних областей файла буде менше кількості кластерів файла, так що обсяг службової адресної інформації в NTFS скорочується порівняно зі схемою адресації файлових систем *ufs/s5*.

4.3. Логічна організація файлової системи

ОС повинна надавати користувачеві зручності під час роботи з даними на дисках. Для цього ОС підміняє фізичну структуру даних зручною для користувача логічною моделлю. При цьому логічна модель файлової системи матеріалізується у вигляді дерева каталогів, символьних складових імен файлів і в командах роботи з файлами.

4.3.1. Типи файлів

Звичайні файли (чи просто файли) містять інформацію довільного характеру, яку заносить у них користувач або яка утворюється в результаті роботи системних і користувальних програм.

Каталоги – це особливий тип файлів, які містять системну довідкову інформацію про набір файлів, згрупованих користувачами за будь-якою ознакою. У каталог можуть входити файли

різних типів, зокрема інші каталоги, за рахунок чого утворюється деревоподібна структура, зручна для пошуку. Каталоги встановлюють відповідність між іменами файлів і їх характеристиками, використовуваними файловою системою для управління файлами.

Спеціальні файли – це файли, асоційовані з пристроями введення / виведення і використовувані для уніфікації механізму доступу до файлів і зовнішніх пристроїв. Спеціальні файли дозволяють користувачеві виконувати операції введення / виведення за допомогою звичайних команд запису у файл або читання з файла. Сучасні ФС підтримують і інші типи файлів (символьні зв'язки, іменовані конвеєри, відображувані в пам'ять файли тощо).

4.3.2. Ієрархічна структура файлової системи

Більшість файлових систем має ієрархічну структуру, в якій рівні створюються за рахунок того, що каталог більш низького рівня може входити в каталог більш високого рівня.

Граф, що описує ієрархію каталогів, може являти собою дерево або мережу. Каталоги утворюють *дерево*, якщо файлу дозволено входити тільки в один каталог, або *мережу*, якщо файл може входити відразу в декілька каталогів. Наприклад, у MS-DOS і Windows каталоги утворюють деревовидну структуру, а в UNIX – мережеву. У деревоподібній структурі кожен файл є листом. Каталог верхнього рівня називається кореневим каталогом, або коренем (*root*).

За такої організації користувач звільнений від запам'ятовування імен усіх файлів; йому досить приблизно уявляти, до якої групи може бути віднесений певний файл, щоб шляхом послідовного перегляду каталогів знайти його. Ієрархічна структура зручна і для багатокористувацької роботи: кожен користувач зі своїми файлами локалізується у своєму каталозі або піддереві каталогів, але разом з тим усі файли в системі логічно пов'язані.

Окремим випадком ієрархічної структури є однорівнева організація, коли всі файли входять в один каталог.

Імена файлів

Усі типи файлів мають символні імена. В ієрархічно організованих файлових системах зазвичай використовуються три типи імен файлів: прості і складні, де останні можуть бути абсолютними і відносними.

Просте, або коротке, символне ім'я ідентифікує файл у межах одного каталогу. Прості імена привласнюють файлам користувачі і програмісти. В ієрархічних файлових системах різним файлам дозволено мати однакові прості символні імена за умови, що вони належать різним каталогам, тобто тут працює схема: «Багато файлів – одне просте ім'я».

Для однозначної ідентифікації файла в таких системах використовується так зване *повне ім'я*, яке являє собою ланцюжок простих символних імен усіх каталогів, через які проходить шлях від кореня до цього файла, плюс просте ім'я самого цього файла.

У деревоподібної файлової системи між файлом і його повним ім'ям маєтись взаємно однозначна відповідність: «Один файл – одне повне ім'я». У файлових же системах, що мають мережеву структуру, файл може входити в декілька каталогів, а отже, мати кілька повних імен; тут справедлива відповідність: «Один файл – багато повних імен». Проте в обох цих випадках файл однозначно ідентифікується повним ім'ям.

Файл може бути також ідентифікований відносним ім'ям, яке визначається через поняття «поточний каталог». При цьому файл ідентифікується ланцюжком імен каталогів, через які проходить маршрут від поточного каталогу до цього файла.

Хоча повне ім'я однозначно визначає файл, операційній системі простіше працювати з файлом, якщо між файлами і їх іменами існує взаємно однозначна відповідність. З цією метою ОС також привласнює файлу унікальне ім'я, так що справедливе співвідношення: «Один файл – одне унікальне ім'я». Унікальне

ім'я існує поряд з одним або декількома символьними іменами, присвоюється файлу користувачами або додатками, являє собою числовий ідентифікатор і призначене тільки для ОС. Наприклад, номер індексного дескриптора в системі UNIX.

Монтування

В обчислювальній системі може бути декілька дискових пристроїв. При цьому один фізичний пристрій за допомогою засобів ОС може бути представлений як декілька логічних пристроїв (зокрема, шляхом розбиття дискового простору на розділи).

Як організувати зберігання файлів у системі, що має декілька пристроїв зовнішньої пам'яті? Для цього можливо два рішення:

1. На кожному з пристроїв розміщується автономна ФС, тобто файли, що містяться на цьому пристрої, описуються деревом каталогів, ніяк не пов'язаним з деревами каталогів на інших пристроях. У такому випадку для однозначної ідентифікації файлу користувач поряд із складовим символьним ім'ям файлу повинен вказувати *ідентифікатор логічного пристрою*. Наприклад, ОС MS-DOS.

2. Користувачеві надається можливість об'єднувати файлові системи, що перебувають на різних пристроях, в єдину ФС, що описується єдиним деревом каталогів. Така операція називається *монтуванням*.

4.3.3. Атрибути файлів

Файл – це не тільки дані, але й атрибути-інформація, що описує властивості файлу. Приклади можливих атрибутів файлу:

- тип файлу (звичайний файл, каталог, спеціальний файл тощо);
- власник файлу;
- творець файлу;
- пароль для доступу до файлу;
- інформація про дозволені операції доступу до файлу;
- час створення, останнього доступу й останньої зміни;

- поточний розмір файла;
- максимальний розмір файла;
- ознака «тільки для читання»;
- ознака «прихований файл»;
- ознака «системний файл»;
- ознака «архівний файл»;
- ознака «двійковий / символний»;
- ознака «тимчасовий» (видаляється після завершення процесу);
- ознака блокування;
- довжина запису у файлі;
- покажчик на ключове поле в записі;
- довжина ключа.

Набір атрибутів файла визначається специфікою файлової системи. Користувач може одержувати доступ до атрибутів, використовуючи засоби, що надані для цих цілей файловою системою. Зазвичай дозволяється читати значення будь-яких атрибутів, а змінювати тільки деякі з них.

Контрольні питання

1. Назвіть способи здійснення введення / виведення. Який з них найкращий? Назвіть його основні переваги.
2. Які програмні рівні введення / виведення?
3. Назвіть основні категорії драйверів пристроїв.
4. Які функції повинен виконувати драйвер пристрою?
5. Що таке файл? Для чого він необхідний?
6. З яких елементів складаються файли простої структури?
7. З яких елементів складаються файли складної структури?
8. Що включає в себе файлова система? Яку роль відіграє файлова система в ОС?
9. Які основні функції файлової системи?
10. Опишіть будову жорсткого диска.
11. Що таке фізичне і логічне форматування диска?

12. Назвіть сучасні, відомі вам файлові системи.
13. Що являє собою файлова система FAT?
14. Які основні можливості файлової системи NTFS?
15. Що таке фізична організація файла? Назвіть основні способи розміщення файла на диску.
16. Які є типи файлів? Опишіть коротко кожен тип.
17. Які є види ієрархічної структури файлової системи? Для яких ОС вони властиві?
18. Які три типи імен файлів використовуються в ієрархічно організованих файлових системах?
19. Що таке монтування? В яких випадках застосовується?
20. Наведіть приклади можливих атрибутів файла.

РОЗДІЛ 5

ТЕНДЕНЦІЇ РОЗВИТКУ СУЧАСНИХ ОПЕРАЦІЙНИХ СИСТЕМ

5.1. Сучасні тенденції в розвитку ОС

Під час стрімкого розвитку сучасних операційних систем усі вони мають відповідати сучасним тенденціям і вимогам, щоб бути конкурентоспроможними на комп'ютерному ринку. Тому розробники ОС мають дотримуватись таких основних вимог.

Підтримка нових мережесих технологій і Web-технологій. Мережі та Інтернет активно розвиваються. З'являються нові стандарти і протоколи: IPv6, HTML 5 (для хмарних обчислень) та ін. Сучасні ОС розвиваються в напрямі підтримки всіх нових мережесих технологій.

Підтримка багатопоточності і багатоядерних процесорів. Усі сучасні ОС мають бібліотеки програм, що підтримують багатопоточність. Саме завдяки багатоядерній архітектурі стає реально можливим паралельне виконання потоків (threads).

Графічні оболонки. Будь-яка сучасна ОС має графічний користувачський інтерфейс, причому (з цілком зрозумілих причин гострої конкуренції між фірмами-розробниками) графічні оболонки для всіх ОС приблизно однакові за можливостями. Іноді користувачеві важко зорієнтуватися, в який саме ОС він працює, хоча для кінцевих користувачів (непрограмістів), мабуть, така уніфікація зручна.

Розвиток файлових систем з метою захисту інформації та значного збільшення розміру файлів (для мультимедіа). Розробляються нові файлові системи, що допускають зберігання досить великих файлів. Іншою вимогою є забезпечення конфіденційності інформації, яке призводить до необхідності реалізації у файлових системах можливості криптування.

Посилена увага до механізмів безпеки і захисту. Багато в чому завдяки ініціативі Trustworthy Computing, розпочатої фірмою Microsoft 2002 р., а також через існування кіберзлочинності всі сучасні ОС приділяють підвищену увагу безпеці: у разі перегляду вебсторінок браузері виконують їх перевірку на відсутність *phishing*; завантаження та інсталяції програм з мережі виконуються тільки з дійсної згоди користувача тощо.

Підтримка розподілених і паралельних обчислень. Сучасні ОС мають у своєму складі високорівневі бібліотеки, що дозволяють розробляти паралельні алгоритми вирішення завдань, наприклад підтримуючі стандарти паралелізму OpenMP і MPI.

Віртуалізація ресурсів і апаратури. Сучасні ОС мають у своєму складі засоби віртуалізації, що дозволяють виконувати додатки для інших платформ в ізольованих віртуальних машинах, в які можуть бути інсталювані інші операційні системи.

5.1.1. Операційні системи різних великих фірм

- IBM – в 1960-х – 1970-х рр. розробила ОС IBM 360/370; потім ОС для персональних комп'ютерів OS / 2; нині найбільш сучасними ОС цієї фірми є z / OS і z / VM;

- Apple – з початку 1980-х рр. розвиває сімейство ОС MacOS, що характеризується поліпшеним графічним інтерфейсом користувача;

- Oracle / Sun – на початку 1980-х рр. фірма Sun розвиває ОС Solaris, розвиток UNIX;

- Hewlett-Packard – розвиває власний діалект UNIX-систему HP / UX;

- Novell – одна з провідних фірм у сфері мережових технологій; розвиває сімейство мережових операційних систем: NetWare; нині Open Enterprise Server (мережева ОС, яка включає всі мережеві можливості NetWare і можливості поширеного діалекту Linux – openSUSE).

Це далеко неповний перелік комерційних і дослідницьких ОС, що включає сотні найменувань.

Solaris – операційна система розробки фірми Oracle / Sun

Вона є розвитком UNIX (лінія Solaris 1.x – розвитком діалекту UNIX BSD, лінія Solaris 2.x – розвитком діалекту UNIX AT & T SVR4). На думку спеціалістів, ця ОС є досить зручною і надійною, а також має зручні інструменти розробки програм. Одна з багатьох особливостей Solaris – це підтримка багатопоточності. Апаратні платформи, підтримувані ОС Solaris: SPARC, x86, IA-32 і x64.

У 1990-х роках багато фірм-користувачів воліли використовувати саме Solaris як серверну ОС, а як сервери – комп'ютери SPARC фірми Sun. Важливою перевагою Solaris є використовуване в цій системі сімейство оптимізуючих компіляторів Sun Studio (C, C++, Fortran). Сьогодні компілятори Sun Studio працюють також на ОС Linux. Досить цікавим і потужним інструментом, доступним у системі Solaris, є відладник розподілених додатків DTrace. Нова файлова система ОС Solaris – ZFS (Zetabyte File System, що підтримує значні обсяги файлів і можливість криптування інформації у файлах з метою її захисту). Система Solaris має потужну підтримку багатопоточності і синхронізації потоків, а також зручну підтримку симетричного багатопроцесування (SMP-архітектур). Використовувані в системі Solaris графічні оболонки: Common Desktop Environment (CDE), GNOME, Java Desktop; у 1990-х рр. використовувалася також графічна оболонка OpenWindows.

MacOS

Розробником MacOS є фірма Apple – законодавець мод у сфері GUI, починаючи з 1980-х рр. Центральною ідеєю MacOS із самого початку є розробка і розвиток ОС тільки на основі графічного інтерфейсу користувача – «ОС без командного рядка».

Апаратна платформа MacOS – різні сімейства комп'ютерів Macintosh фірми Apple (найбільш популярні серед робочих станцій у США), а також PowerPC – робоча станція RISC-архітектури, спільно розроблена Apple, IBM і HP.

Версії MacOS розрізняються за своїм підходом до реалізації, хоча для користувача, завдяки зручному графічному інтерфейсу, ці відмінності можуть бути непомітні. Класична MacOS (classic MacOS) – оригінальна розробка фірми Apple; нова лінія MacOSX – розвиток ОС MacOS Classic і ОС NeXTSTEP (UNIX-подібної ОС), тобто вона є UNIX-сумісною.

HP-UX

Розробником ОС HP-UX є компанія Hewlett-Packard. Це UNIX-подібна 64-бітова ОС (одна з перших 64-бітових ОС в історії). Її апаратна платформа – робочі станції фірми HP. У системі HP-UX використовується оригінальна віртуальна файлова система: Veritas File System (Vx-Fs) використовує представлення файлів на основі розширень (extents).

5.1.2. Переваги 64-розрядної версії Windows

Ще декілька років потому на більшості домашніх комп'ютерів працювала 32-бітова версія Windows, а 64-бітний варіант використовували виключно професіонали (наприклад, для ресурсоемних програм відеомонтажу). При цьому процесори із 64-розрядної архітектурою випускаються вже з 2003 року, а двома роками пізніше на ринок вийшла Windows XP (не підтримується розробниками) в 64-бітному варіанті. Але на більшості нових комп'ютерів 64-бітові версії Windows почали встановлювати тільки після появи Windows 7.

Переваги 64-розрядної системи

Збільшена робоча швидкість. З 64-бітовою версією Windows комп'ютер може обробляти за одиницю часу в два рази більше даних, ніж з 32-бітовою. При цьому він використовує розширені можливості 64-розрядних процесорів, які за один такт здатні обробляти 64 біта даних (8 байт). Тому програми, оптимізовані для 64-бітових ОС, вміють працювати швидше, ніж їх аналоги для процесорів, що обробляють усього 32 біта (4 байти) за один такт.

Більше оперативної пам'яті. 32-бітові версії Windows можуть використовувати максимум 4 Гб пам'яті, причому не вся вона доступна програмам. Наприклад, 1 Гб займає сама операційна система, до того ж до 1 Гб (залежно від ПК) резервується для відеопам'яті. Тож для самих програм залишається не більше 2 Гб. Якщо вони «ненажерливі», пам'яті може не вистачити. Для 64-бітових версій Windows подібного обмеження не має. Максимальна робоча пам'ять у Windows 7 Home Premium зросла до 16 Гб; версії Professional, Ultimate і Enterprise можуть працювати навіть з 192 Гб. Завдяки більш доступній пам'яті Windows рідше доводиться вивантажувати розділи програм на порівняно повільний жорсткий диск (у файл підкачки), тому і комп'ютер працює швидше.

Вимоги до комп'ютера

Процесор. Найважливішою необхідною умовою для роботи 64-бітної операційної системи є процесор із 64-бітною архітектурою. Виробник процесорів AMD називає цю технологію AMD64, а його конкурент Intel – EM64T.

Процесори із 64-бітною архітектурою вже стали швидше правилом, ніж винятком. Приблизно з 2006 року більшість нових комп'ютерів і ноутбуків оснащується саме ними, хоча багато ще працюють під 32-бітною Windows. Дізнатися, який саме процесор встановлений на вашому ПК, можна за допомогою програми CPU-Z.

Оперативна пам'ять. 64-бітна Windows має сенс лише в тому випадку, якщо у комп'ютера не менше 4 Гб оперативної пам'яті. В іншій 64-бітній версії ОС не пред'являють особливих вимог до апаратної частини (наприклад, до HDD або відеокарти) порівняно з 32-бітною.

Якщо в 64-бітній Windows працюють 32-бітові програми, то в диспетчері завдань вони позначаються відповідним способом (* 32) (рис. 5.1). Диспетчер завдань можна запустити, натиснувши комбінацію клавіш Ctrl + Shift + Esc.

Имя образа	Пользо...	ЦП	Память (частный рабо...	Описание
atiedxx.exe		00	2 240 КБ	
avp.exe *32	Эльдар	00	3 000 КБ	Kaspersky Anti-Virus
CCC.exe	Эльдар	00	24 840 КБ	Catalyst Control Center: Hos...
csrss.exe		00	2 060 КБ	
DAODx.exe *32	Эльдар	00	332 КБ	DAODx
dmaster.exe *32	Эльдар	00	9 780 КБ	Download Master

Рис. 5.1. Диспетчер завдань

Як дізнатися версію операційної системи? Це надто просто: клацніть по кнопці «Пуск», потім правою кнопкою миші – по «Комп'ютер» і виберіть «Властивості». У вікні навпроти «Тип системи» побачите версію системи.

Недоліки 64-розрядної системи

Потрібні нові драйвери. Усі пристрої вимагають для 64-бітної Windows відповідних драйверів (аналоги для 32-бітної ОС працювати не будуть). Для сучасних пристроїв це, звичайно, не проблема, але для застарілих моделей багато виробників не розробляють 64-бітові версії драйверів. Тому перш ніж переходити на 64-бітову Windows, зайдіть на вебсторінку виробника пристрою і переконайтеся, що для вашого обладнання доступна 64-бітова версія драйвера.

Потрібно більше пам'яті. Багато структури даних у 64-бітових програмах мають розмір 8 байт (64 біт). Тому програми займають на 10–20 % більше місця на жорстких дисках, ніж відповідні 32-бітні версії з 4-байтними структурами. Потреба в оперативній пам'яті для 64-бітного програмного забезпечення також підвищується приблизно на 15 %.

Краща п'ятірка програм для 64-бітної системи

Ці програми чудово працюють у 64-бітній операційній системі і користуються підвищеним попитом у користувачів.

7-ZIP. Програма-архіватор, призначена для стиснення файлів. Може бути корисна, наприклад, коли потрібно скопіювати багато великих файлів на USB-драйв.

Nitro PDF Reader. Ця проста програма дозволяє легко створювати, переглядати та редагувати документи PDF-формату.

nHancer. Утиліта забезпечує доступ до налаштувань графічної плати NVIDIA. Працює з усіма поширеними моделями відеокарт і навіть з SLI-конфігураціями. Можна вибирати профілі налаштувань для різних ігор.

ProgDVB. Якщо у ПК встановлено цифровий ТВ-тюнер, за допомогою ProgDVB є можливість переглядати телевізійні передачі і записувати їх на жорсткий диск. Програма працює з ТВ-тюнерами різних виробників. Крім безкоштовного варіанта, є розширена платна версія.

Blender. Програма для роботи з тривимірними об'єктами. До її складу входять засоби для моделювання, анімації, постобробки відео, а також створення інтерактивних ігор.

Виникає питання, чи працюють 32-бітові програми під 64-бітною Windows? Підсистема перетворення під назвою Windows-on-Windows 64-bit (WoW64) забезпечує спільну роботу 32-бітного програмного забезпечення в 64-бітній операційній системі. За її допомогою більшість 32-бітних програм працює бездоганно але, на жаль, є винятки. Труднощі зазвичай виникають з програмами таких категорій:

- *резервне копіювання.* Програми для створення бекапів безпосередньо працюють із системними папками Windows. Але WoW64 іноді перенаправляє доступ 32-бітного програмного забезпечення в інші папки. Якщо програма цього не враховує, резервне копіювання може виявитися неповним або навіть помилковим;

- *оптимізація Windows.* Старі, розраховані на 32-бітову Windows програми оптимізації не можна запускати на 64-бітній Windows. Часом вони здатні більше нашкодити, ніж допомогти;

- *антивіруси.* Якщо потрібен надійний захист для ПК, використовуйте в 64-бітній Windows тільки 64-бітові версії антивірусів і аналогічних програм. Застаріле 32-бітне захисне програмне забезпечення нездатне контролювати всі системні папки, а отже, може пропустити шкідників, розроблених спеціально

для атак на 64-бітову Windows. Сучасні антивірусні пакети відомих виробників, наприклад «Лабораторії Касперського», Norton або Symantec та інші, підходять як для 32-, так і для 64-бітної Windows.

Якщо програма під 64-бітову Windows працює зі збоями, позбавитися від них нерідко допомагає установка новітньої версії і всіх оновлень з вебсайту виробника. До того ж у «Центрі сумісності Windows» можна перевірити, чи сумісні ваші пристрої і програми з 64-бітною системою.

У 64-бітній Windows програми працюють швидше. Деякі 64-бітові програми були всього лише перетворені для застосування в 64-бітній Windows, але, по суті, залишилися 32-бітними. Наприклад, хоча вони і мають 64-бітні структури даних, однак зберігають у них, як і раніше, 32 біта даних. Подібні програми, як правило, не мають змоги працювати швидше. 64-бітові програми реалізують свої переваги, лише якщо з самого початку розроблялися й оптимізували для застосування в 64-бітових Windows. У тесті ComputerBild такі програми в середньому працювали на 13 % швидше 32-бітних «побратимів». Старе 32-бітне програмне забезпечення в 64-бітній Windows працює навіть повільніше, ніж у 32-бітній системі; причина – перетворення підсистемою WoW64 64-бітових програм. Знайти подібні програми досі не так легко, як хотілося б. У таблиці 5.1 приведені приклади 64-бітових програм, багато з них безкоштовні і без особливих зусиль скачуються з сайту розробника.

Таблиця 5.1

Приклади 64-бітових програм

Програма	Версія	Категорія	Інтернет-адреса
1	2	3	4
7-Zip	9.20	Архіватор	www.7-zip.org.ua/ru
AVG Anti-Virus Free	2011	Антивірусна програма	www.freeavg.com
Blender	2.59	Редагування 3D-моделей	www.blender.org

Продовження таблиці 5.1

1	2	3	4
CCleaner	3.10	Очистка системи	www.piriform.com
Defraggler	2.06	Дефрагментатор	www.piriform.com
GIMP	2.7.3	Обробка зображень	www.gimp.org
Image Composite Editor	1.4.4	Створення панорам	www.microsoft.ru
Internet Explorer	9	Інтернет-браузер	www.microsoft.ru
iTunes	10.4	Відтворення музики і відео	www.apple.com/ru/itunes
Mathematics	4.0	Математична програма	www.microsoft.ru
MediaCoder	2011 R8	Конвертоване відео	www.mediacoderhq.com
Mixxx	1.9.0	DJ-програма	www.mixxx.org
nHancer	2.5.9	Оптимізація графічних плат	www.nhancer.com
Nitro PDF Reader	2.0	Редагування PDF-файлів	www.nitroreader.com
Office	2010	Офісний пакет	www.microsoft.ru
Paint.NET	3.5.8	Обробка зображень	www.getpaint.net
Partition Manager	11	Управління розділами жорстких дисків	www.paragon.ru
PDF Xchange Viewer	2.5.197	Перегляд PDF	www.tracker-software.com
ProgDVB	6.70.6	Перегляд і запис ТВ-передач	www.progdvb.com
SafeErase	5.0	Надійне видалення даних	www.oo-software.com
Sandboxie	3.58	Захищене оточення	www.sandboxie.com
Total Commander	7.56a	Файловий менеджер	www.wincmd.ru
SyncToy	2.1	Синхронізація	www.microsoft.ru
Thunderbird	6.0.1	Поштова програма	www.mozilla.org/ru/thunderbird
Vegas Pro	10	Програма відеомонтажу	www.sonycreativesoftware.com
Virtual Dub	1.9.11	Конвертування відео	www.virtualdub.org
Virtual PC	6.1	Програма віртуалізації	www.microsoft.ru
WinRAR	4.01	Архіватор	www.rarlab.com

5.1.3. Windows 8 як новий етап у розвитку операційних систем Microsoft

У Windows 8 було приділено увагу системі розпізнавання голосу і голосовому управлінню. Серед інших можливих нововведень була поліпшена робота віртуалізації, що дозволяє віртуалізувати всю операційну систему; тісної інтеграції з мобільною версією операційної системи, підтримка розпізнавання жестів. У Windows 8 удосконалена система глибокого сну та відновлення, а також нові функції забезпечення безпеки, включаючи модифікацію PatchGuard.

Згідно з презентацією Microsoft у Windows 8 був істотно скорочений час завантаження і вимкнення порівняно з попередніми ОС сімейства Windows. Приділена увага енергоспоживанню та енергоефективності. Заявлена підтримка інтерфейсу USB 3.0, з'єднань Bluetooth 3.0, а також бездротових і стереоскопічних дисплеїв.

Одним з нововведень на той час була система розпізнавання особи користувача за допомогою вебкамери. Ця система може самостійно переводити ПК у різні режими енергоспоживання залежно від того, чи перебуває користувач перед вебкамерою, чи ні. У певному вигляді подібна концепція реалізована в ігровому контролері Kinect для ігрової консолі Xbox 360.

У Windows 8 уже присутні нові інструменти для забезпечення безпеки, які спрощують діагностику та усунення несправностей. Є функція «скидання системи», яка дозволить перевстановити Windows 8 із збереженням при цьому всіх файлів користувача. Також Windows 8 підтримує акселерометри, шифрування жорсткого диска, підтримку GPS у нетбуках і ноутбуках, використання WWAN-модулів.

Особливості Windows 8

1. Усі традиційні програми для Windows 7 працюватимуть і під Windows 8.

2. Windows 8 має такі самі системні вимоги, як і попередня версія операційної системи: процесор частотою 1 ГГц, 1 ГБ оперативної пам'яті і не менше 16 ГБ вільного дискового простору.

3. Windows 8 використовує новий механізм виключення під назвою Hybrid Boot, який прискорює завантаження і вимикання системи в декілька разів. Технологія являє собою гібрид традиційного «холодного» завантаження і відновлення роботи після глибокого сну.

4. Антивірус включений у систему за замовчуванням. Крім того, у Windows 8 є поліпшені засоби запобігання атак, деякі з яких вбудовані в ядро системи.

5. Windows 8 підтримує функцію Near Field Communication, яка дозволяє різним цифровим пристроям взаємодіяти один з одним або, наприклад, зі спеціальними позначками за допомогою простих торкань.

6. У Windows 8 є вбудована підтримка технології віртуалізації Hyper-V. Це дозволяє одночасно запускати декілька 32-розрядних або 64-розрядних операційних систем x86 на одному комп'ютері.

7. За допомогою функції Portable Workspace користувач може зробити клон операційної системи на USB-носії. Обсяг носія повинен бути не менше 16 ГБ.

8. Додані функції оновлення та обнулення стану системи. Refresh використовується в тому випадку, якщо комп'ютер став працювати нестабільно і дозволяє оновити його стан без втрати даних користувача і додатків. Reset можна використовувати для приведення системи в первинний стан на момент установки.

9. Для входу в систему можуть використовуватися біометричні дані або розпізнавання осіб.

10. Система має розширену підтримку роботи з декількома моніторами. Тепер можна відображати панель завдань на будь-якому моніторі, керувати шпалерами на різних екранах, відображати з одного боку стартове меню, а з іншого – робочий стіл тощо.

5.1.4. Windows 10

Після випуску операційної системи Windows 8 вважається, що пропуск номера «9» не був пов'язаний із сумісністю. У компанії хотіли, щоб користувач не порівнював нову версію операційної системи з невдалою Windows 8.

Розроблена і представлена Windows 10 – це нова операційна система сімейства Windows, яка створена універсальною для таких гаджетів, як ПК, смартфони, планшети, консолі Xbox та інші, з єдиними платформою розробки і магазином оригінальних додатків.

З 1 червня 2015 року користувачі Windows 7 і 8.1 змогли зареєструвати заявку на безкоштовне оновлення на нову версію, тобто на Windows 10. До того ж, це могли зробити як офіційні власники ліцензованого ОС, так і «пірати».

Нова операційна система вийшла у липні та одночасно з'явилася у продажу у 190 країнах, а також стали доступні мобільні версії та адаптації під інші гаджети.

Що стосується найпримітніших оновлень, то у Windows 10 повернулася кнопка «Пуск», а також з'явилася система біометричної ідентифікації Windows Hello. Її завданням є вхід у систему без пароля на мобільних пристроях.

ОС Windows 10 отримала інтелектуальний голосовий помічник Cortana, управління яким здійснюється голосом і текстовими командами. Сервіс аналізує запити користувача, вибиваючи найоптимальніші результати.

У Windows 10 також з'явився новий інтернет-браузер Microsoft Edge, що прийшов на зміну Internet Explorer. Крім того, з випуском Windows 10 компанія Microsoft перейшла на нову бізнес-модель SaaS (Windows просувається як сервіс). Тобто всі оновлення, актуальні параметри безпеки та сервіси встановлюються в автоматичному режимі.

Системні вимоги:

1. *Процесор*: не менше 1 ГГц або SoC.
2. *ОЗП*: 1 ГБ (для 32-розрядних систем) або 2 ГБ (для 64-розрядних систем).
3. *Місце на жорсткому диску*: 16 ГБ (для 32-розрядних систем) або 20 ГБ (для 64-розрядних систем).
4. *Відеокарта*: DirectX версії 9 або вище з драйвером WDDM 1.0.
5. *Дисплей*: 800 x 600.

У таблиці 5.2 наведено переваги Windows 10.

Таблиця 5.2

Переваги і недоліки Windows 10

ПЕРЕВАГИ	НЕДОЛІКИ
Новий інтерфейс значків	Обмежена безкоштовність
Оновлене меню «Пуск»	Не можна відключити оновлення
Оновлення: в ОС Windows 10 включено автоматичне завантаження та встановлення оновлень. Тепер відмовитися від них уже буде неможливо	Несумісність більшості ігор з новою ОС
Нова система безпеки, тобто паролі вже не підходять для входу в ПК, тепер це перевірка відбитка пальця, оболонки ока, а також рис обличчя	З приводу паролів: такі паролі діють лише за наявності спеціального обладнання
Стабільна робота: ОС Windows 10 працює набагато швидше, збільшилася кількість драйверів	Відсутність медіаплеєра Windows Media Center
Браузер Microsoft Edge – новий, універсальний та інтелектуальний браузер, створений спеціально для ОС Windows 10	Прискорена розрядка деяких ПК (ноутбуків)
Інтелектуальні функції: Cortana – віртуальний помічник, який розрізняє ваші вирази «на слух»	Помічник Cortana не підтримує російську та українську мови, і в наступних оновленнях ОС не поставлена мета це змінити

Windows 10 є першою операційною системою компанії Білла Гейтса, яка офіційно стала поширюватися із серверів постачальника, і з ПК її користувачів заснована на принципі протоколу BitTorrent.

Компанія Microsoft стверджувала, що Windows 10 стане останнім продуктом сімейства Windows.

5.1.5. ОС для хмарних обчислень (cloud computing).

Windows Azure

Хмарні обчислення засновані на масштабованих і віртуалізованих ресурсах (даних і програмах), які доступні користувачам через Інтернет і реалізуються на базі потужних центрів обробки даних (data centers). Виділяють доступні «хмари», що надаються

різними компаніями, які можна використовувати для доступу до потужних обчислювальних ресурсів, а користувач сплачує абонентську плату за використання хмарних сервісів певної фірми.

«Хмарний» підхід має певні недоліки: по-перше, є повна залежність користувача від використовуваної ним хмари, оскільки через хмару доступні не тільки програми, але і дані самого користувача; по-друге, виникає багато питань щодо безпеки даних користувача, які зберігаються в хмарі.

З хмарних платформ найбільш популярною є Microsoft Windows Azure – операційна система з підтримання хмарних обчислень. І Microsoft Azure Services Platform – платформа для розробки і використання хмарних сервісів на базі Microsoft.NET.

Сьогодні багато великих компаній: Microsoft, Google, IBM, Oracle/Sun, Amazon – займаються розробкою своїх хмарних сервісів та інструментів для їх створення. Є тенденція до інтеграції «корпоративних хмар» в єдину доступну користувачеві хмару.

Концепція хмарних обчислень базується на таких положеннях: інфраструктура як сервіс, платформа як сервіс, програмне забезпечення як сервіс, а також бізнес-додатки, доступні через Інтернет. Іншими словами, організація хмарних обчислень докорінно змінює архітектуру системи: в ній необхідно представити всі можливості обробки даних, використання програм, налаштування тощо як хмарні сервіси.

Розрізняються такі рівні архітектури хмарних обчислень.

Рівень клієнта – це клієнтське програмне забезпечення, що використовується для доступу до хмарних сервісів, наприклад веббраузер.

Рівень сервісів – це самі сервіси, використовувані через хмарну модель.

Рівень додатків – це програми, доступні через хмару і не потребують інсталяції на комп'ютері користувача (в принципі одна з основних переваг хмарної моделі).

Рівень платформи – це програмна платформа, яка об'єднує повний набір інструментів для розгортання та використання хмарних обчислень на комп'ютері користувача (без додаткових

інсталяцій, купівлі обладнання та ін.). Приклад такої платформи: Microsoft.NET Azure Services Platform.

Рівень пам'яті – підтримка зберігання даних користувача і доступу до них через хмару.

Рівень інфраструктури – надання повної віртуалізованої платформи через хмару, наприклад Amazon EC2.

Наведемо схему архітектури хмарних обчислень:

- **сервіси**, доступні через хмару;
- **інфраструктура** для їх розгортання і використання;
- **платформа** – набір інструментів для використання хмари;
- **пам'ять** – підтримання зберігання призначених для користувача даних у ЦОД, що реалізує хмару;
- **архітектор хмари** – це основний розробник його архітектури;
- **інтегратор хмари** – це його системний адміністратор, що відповідає за додавання компонентів у хмару і їх зміну;
- **компоненти хмари** – як правило, є вебсервісами.

Хмара може бути загальнодоступною або приватною (корпоративною).

Під час використання хмарних обчислень дещо змінюються і ролі фахівців, що беруть участь у них. Постачальником хмари є центр обробки даних. Користувачами хмари можуть бути будь-які користувачі Інтернету. Виробник устаткування або програмного забезпечення – це компанія, що забезпечує розробку апаратури і програмного забезпечення для центру обробки даних. Модель хмарних обчислень заснована на дотриманні ряду стандартів:

- стандарти HTTP (основний вебпротокол) – для взаємодії додатків;
- XMPP (Jabber) – стандарт для відправки й отримання миттєвих повідомлень;
- SSL (Secure Socket Layer) – рівень безпечних сокетних мережеских з'єднань.

Для роботи клієнтів у хмарі використовуються веббраузери (з активним використанням технології AJAX, що дозволяє зменшити кількість перенаправлень з однієї вебсторінки на іншу і тим

самим час доступу користувача до необхідної йому інформації) та офлайн-клієнти, робота яких заснована на HTML 5 (спеціальній версії HTML для хмарних обчислень).

Для реалізації хмари використовуються принципи віртуалізації програм і даних та стандарт OMF. Для взаємодії з сервісами дані передаються у форматі XML.

Windows Azure

Windows Azure – хмарна платформа, розроблена фірмою Microsoft (по суті, операційна система і набір інструментів «у хмарі»). На рисунку 5.2 ілюструється роль Windows Azure в організації використання хмарних сервісів, що надаються центром обробки даних Microsoft, фірмами та індивідуальними замовниками.



Рис. 5.2. Windows Azure, користувачі та центри обробки даних

Зауважимо, що Windows Azure забезпечує зберігання, використання та модифікацію даних і запуск програм тільки на комп'ютерах центрів обробки даних Microsoft. Ніякого програмного забезпечення, крім інтернет-браузера, на користувацьких комп'ютерах не потрібно.

Перспективи хмарних обчислень, незважаючи на їх критику і досі обережне ставлення до них, надто великі, оскільки хмарні обчислення дозволяють отримати доступ до потужних обчислювальних можливостей і даних значного обсягу практично з будь-

яких комп'ютерів користувачів (зокрема, кишенькових, мобільних пристроїв тощо), що мають веббраузер.

5.2. Особливості ОС для мобільних пристроїв

До мобільних пристроїв прийнято відносити мобільні телефони, смартфони і комунікатори. Розробники ОС для мобільних пристроїв працюють над тим, щоб наблизити можливості цих ОС до можливостей ОС для настільних і портативних комп'ютерів. Однак в ОС для мобільних пристроїв є своя специфіка, яку необхідно враховувати під час розробки таких ОС.

1. Необхідно враховувати *більш жорсткі обмеження щодо пам'яті* мобільних пристроїв. Хоча мобільні пристрої активно розвиваються, за своїми параметрами (обсягом пам'яті, швидкодії процесора) вони все ще досі поступаються настільним комп'ютерам. Тому додатки для мобільних пристроїв, які потребують значного обсягу пам'яті, відтворюються на мобільних пристроях з неповними можливостями. Ряд інструментів, наприклад Java, також доступні для мобільних пристроїв у спеціальних версіях, розроблених з метою економії пам'яті, з обмеженнями, не властивими класичним версіями: на мобільних пристроях працює Java Micro Edition (JME), а не повна версія Java Standard Edition (JSE) для настільних комп'ютерів. У ній, наприклад, відсутня дійсна арифметика в Java і ряд інших важливих можливостей.

2. Необхідно враховувати *незначну швидкість процесора*. Порівняно з настільними комп'ютерами аналоги настільних додатків на мобільних пристроях працюють помітно повільніше: наприклад, повільно відкривається зображення на екрані мобільного телефону, текстовий файл для перегляду тощо. Це незручно для користувачів і, мабуть, буде подолано в наступних версіях самих мобільних пристроїв і їх ОС.

3. Необхідно враховувати *особливості екранів та екранних навігаторів* конкретних моделей мобільних пристроїв. Багато типів мобільних пристроїв різних фірм мають суттєво різні екрани і різні види екранних навігаторів (трекбол та ін.) Під час розробки

ОС і сервісних програм для мобільних пристроїв ці відмінності доводиться враховувати, що ускладнює розробку програмного забезпечення.

4. Необхідно враховувати *сумісність з основними форматами файлів*: .doc / docx, .ppt / .pptx, .pdf, .jpg та ін. Під час роботи на мобільному пристрої потрібно мати можливість візуалізувати, редагувати і створювати файли тих самих звичних форматів, що і на настільних комп'ютерах. Забезпечення такої сумісності – одне з важливих завдань ОС і сервісних програм для мобільних пристроїв.

5. *Мультимедійні можливості*: малюнки, відео, аудіо, обмін мультимедійними повідомленнями. Обробка мультимедійної інформації для мобільних пристроїв особливо важлива: користувачі повинні мати можливість зробити фотографії, зняти відео, переглянути їх на мобільному пристрої, опублікувати в Інтернеті, послати на настільний комп'ютер або у вигляді повідомлення своїм кореспондентам.

6. *Підтримка комунікаційних і мережних технологій*: Wi-Fi / WiMAX, Bluetooth, GPRS, EVDO, GSM, CDMA. Мобільний пристрій і його ОС повинні забезпечувати для користувачів можливість сучасних видів комунікації у бездротових мережах.

5.2.1. Короткий огляд ОС для мобільних пристроїв

Сьогодні на ринку мобільних пристроїв використовується декілька десятків найбільш поширених ОС. Деякі з них розроблені на основі вільно поширюваного ядра Linux. Провідні фірми-виробники мобільних пристроїв підтримують власні ОС або ОС, придбані разом з їх фірмами-розробниками. Найбільш поширені ОС для мобільних пристроїв такі:

- Nokia Symbian OS;
- Google Android;
- Windows Mobile;
- Blackberry OS;
- Apple iPhone OS;
- Samsung Bada;
- PalmOS.

Windows Mobile (з 2011 року не розробляється і не підтримується).

Windows Mobile – сімейство ОС для мобільних пристроїв фірми Microsoft. Воно належить до сімейства Windows CE (Consumer and Embedded) – Windows для вбудованих систем. Ядро ОС Windows Mobile заснована на ОС Windows CE.

У США Windows Mobile – третя за популярністю ОС для мобільних пристроїв (після BlackBerry OS і iPhone OS). Windows Mobile підтримувала такі види мобільних пристроїв: PocketPC, смартфони, комунікатори (наприклад, Qtek). Перша версія Windows Mobile була випущена 1996 р.

Windows Mobile надавала різноманітний набір можливостей і програмне забезпечення:

- **Office Mobile** – аналог Microsoft Office для мобільних пристроїв; повна сумісність за форматами;
- **Windows Media Player** – мультимедійний програвач, аналог програвача для настільної версії Windows;
- **Internet Explorer Mobile** – веббраузер, аналог Internet Explorer для настільної версії Windows;
- програмне забезпечення для підтримки **Bluetooth і Wi-Fi** – сучасних видів комунікації;
- програмне забезпечення **Microsoft ActiveSync** для синхронізації даних з настільними комп'ютерами.

Windows Mobile підтримувала користувацький інтерфейс з мобільним пристроєм за допомогою торкання екрана стилусом і пальцями, зокрема (в сучасних версіях) multi-touch.

Останні версії Windows Mobile підтримували також .NET Compact Framework, що давало можливість виконання додатків для платформи .NET на мобільних пристроях.

Версії Windows Mobile 6 (з 2011 року не розробляється і не підтримується). Починаючи з версії 6, Windows Mobile мала три основні версії:

- Windows Mobile 6 Classic – для КПК;
- Windows Mobile 6 Professional – для комунікаторів;
- Windows Mobile 6 Standard – для смартфонів.

Symbian OS – найбільш поширена в певний час ОС для мобільних пристроїв, розроблена консорціумом Symbian (Nokia, Ericsson, Psion, Motorola), що заснований 1998 р. Фірма Nokia фінансувала об'єднання Symbian Foundation, метою якого була розробка і підтримка єдиної (для мобільних пристроїв різних компаній) мобільної платформи на основі Symbian OS. Symbian OS розроблена на основі ОС Psion EPOC32 (фірми Psion). Мова реалізації системи – C++; є також підтримка Java. Найбільш поширені версії: Symbian OS Series 60 2nd edition; 3rd edition. Наприкінці 2012 року закrywся вебсайт «symbian.nokia.com».

Google Android – стек додатків для мобільних пристроїв, який включає операційну систему (на базі ядра Linux), проміжне програмне забезпечення (middleware) і сервісні програми. Система Android розроблена фірмою Android, Inc., яка придбана компанією Google (2005). Важливою особливістю Google Android є те, що сервісні програми та бібліотеки цієї системи написані на Java.

Можливості Google Android. Google Android має зручний та естетичний інтерфейс користувача, який розроблений з використанням двовимірної і тривимірної графіки (бібліотеки OpenGL). Основні можливості системи такі:

- СУБД SQLite для зберігання даних;
- підтримувані мережеві технології: GSM / EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, WiMAX;
- обмін повідомленнями SMS та MMS;
- веббраузер на базі WebKit Application Framework.

Підтримка Java. Фірма Google з принципових міркувань використовує в системі Android власну реалізацію Java – Dalvik Virtual Machine, розроблену спеціально для мобільних пристроїв. На думку фахівців Google, стандарт Java Micro Edition (JME) застарів, тому що розрахований на застарілі типи мобільних пристроїв і їх технічні можливості. Тому в Google Android стандарт JME не підтримується.

Підтримка мультимедіа. У системі Google Android є кодеки для всіх поширених мультимедійних стандартів, програмне забезпечення для обробки мультимедійних файлів і взаємодії з відео- і аудіопристроями.

Підтримка розробки додатків. Система Google Android має власне інтегроване середовище для розробки додатків – Android SDK, що включає емулятор мобільних пристроїв, засоби налагодження, профілювання, а також plug-in до популярного середовища Eclipse для розробки Java-додатків.

Перспективи розвитку ОС для мобільних пристроїв

У зв'язку з активним розвитком мобільних пристроїв, операційні системи для них мають значні перспективи розвитку за такими напрямками:

- покращення та спрощення інтерфейсу користувача;
- покращена графіка;
- більш широкі мультимедійні можливості;
- розвиток набору сервісних та ігрових програм;
- забезпечення повної сумісності з настільними комп'ютерами і використовуваними на них форматами файлів;
- продовження і розвиток використання платформи Java для мобільних пристроїв; усі провідні виробники мобільних пристроїв підтримують платформу Java, що є гарантією розвитку самої Java-технології;
- розвиток самих мобільних пристроїв: прискорення процесорів, збільшення обсягу пам'яті, реалізація нових швидких комунікаційних технологій і підтримання цих нових можливостей в ОС для мобільних пристроїв.

5.2.2. Розквіт ОС у 2000-х роках

Нині ми є свідками небувалого розквіту операційних систем.

Безперечним лідером у цій галузі є корпорація Microsoft, що випустила менш ніж за 10 недавніх років серію ОС сімейства Windows: Windows XP, Windows 2003, Windows Vista (2007), Windows 2008, Windows 2008 High-Performance Computing (HPC), Windows 7, Windows 8, Windows 10 з постійними автоматичними оновленнями.

Розвиваються також діалекти ОС Linux (Red Hat, Fedora, Mandrake, Ubuntu, SuSE та ін. сотні діалектів). Linux – операційна

система типу UNIX, ядро якої вільно поширюється з вихідними кодами.

Фірма Sun (2010 р. увійшла до складу фірми Oracle) розробляє і випускає ОС Solaris – одну з найбільш сучасних ОС типу UNIX з розвиненою підтримкою паралельного програмування, новими видами файлових систем, що відрізняються своєю підвищеною надійністю.

Існує також багато інших операційних систем. У США і Канаді, як відомо, найбільш популярні комп'ютери сімейства Macintosh фірми Apple (Mac) зі своєю операційною системою MacOS, яка є законодавцем мод у сфері графічних користувацьких інтерфейсів (GUI) і обміну мультимедійною інформацією (наприклад, мовного введення). Назвемо також ОС фірми IBM для суперкомп'ютерів і комп'ютерів загального призначення (mainframes).

Особливо важливо те, що сьогодні багато з них (або їх великі частини, наприклад ядро) доступні з відкритим вихідним кодом.

Корпорація Microsoft поклала початок цьому руху 2003 р., коли була оголошена академічна програма Windows Embedded Shared Source, – був відкритий вихідний код Windows для вбудованих систем. А 2006 р. відбулася і зовсім неймовірна досі у світі програмістів подія: Microsoft відкрила «святую святих» – вихідний код ядра ОС Windows сімейства NT (NT/2000/XP/2003/2008/7) і надала в розпорядження університетів та академічних організацій Windows Research Kernel (WRK) – самодокументований вихідний код «дослідницького» ядра Windows. Тепер кожен студент, викладач і дослідник мають можливість вивчати систему Windows «зсередини» і навіть розвивати її, але тільки для цілей навчання і досліджень, а не для комерції.

Фірма Sun (нині Oracle) поклала початок аналогічній ініціативи для ОС Solaris – декілька років потому було започатковано проєкт OpenSolaris. Результати цього академічного проєкту використовуються у разі випуску нових версій комерційної ОС Solaris.

За традицією, ще з початку 1990-х рр., ядро ОС Linux також поширюється вільно, з вихідними кодами, що викликало хвилю

робіт зі створення нових діалектів Linux, а також з розробки нових ОС для мобільних пристроїв на базі ядра Linux (наприклад, ОС Google Android).

Також інтенсивно розвиваються ОС для мобільних пристроїв. Ще декілька років потому найбільш використовуваними ОС у цій сфері були ОС сімейства Symbian. Однак нині ОС Microsoft Windows Mobile і Google Android активно тіснять Symbian з ринку.

ОС для хмарних обчислень – принципово новий вид ОС, що відображає сучасну тенденцію до організації обчислень, таких як хмарних (cloud computing). Хмара – це метафора Інтернету. У разі хмарних обчислень користувач зі свого комп'ютера одержує платний доступ через Інтернет до вебсервісів, які працюють на комп'ютерах потужних центрів обробки даних (наприклад, на серверах Microsoft). При цьому не тільки використовуване програмне забезпечення (у вигляді набору вебсервісів), а й самі оброблювані дані користувача зберігаються на серверах «хмарного» центру обробки даних. На своєму комп'ютері користувач має лише простий, зручний «хмарний» вебінтерфейс, що не вимагає значних ресурсів. Найбільш поширеною ОС для хмарних обчислень є сьогодні Microsoft Windows Azure.

Контрольні питання

1. У чому полягає вимога до сучасних ОС щодо підтримання багатопоточності і багатоядерних процесорів?
2. Які основні вимоги висуваються під час розробки сучасних ОС?
3. Які переваги має ОС Solaris?
4. У чому основні переваги ОС MacOS? Яка фірма розробник цієї ОС?
5. Назвіть основні переваги 64-розрядних систем.
6. Які основні вимоги до комп'ютера у разі використання 64-бітної операційної системи?
7. У чому основні недоліки 64-розрядної системи?
8. Чи працюють 32-бітові програми під 64-бітну Windows?

9. Які ви знаєте найважливіші 64-бітові програми?
10. Чому Windows 8 вважається новим етапом у розвитку операційних систем Microsoft?
11. Що являє собою Windows Research Kernel? З якою метою використовується?
12. На чому засновані хмарні обчислення?
13. У чому недоліки «хмарного» підходу до ОС?
14. На яких положеннях базується концепція хмарних обчислень?
15. Які є рівні архітектури хмарних обчислень?
16. Що являє собою хмарна платформа Windows Azure?
17. Що необхідно враховувати під час розробки ОС для мобільних пристроїв?
18. Назвіть найбільш поширені ОС для мобільних пристроїв? Опишіть найвдалішу на вашу думку.
19. Які основні різновиди мобільних пристроїв?
20. Які основні особливості ОС для мобільних пристроїв?
21. З якими основними форматами файлів необхідно забезпечити сумісність в ОС для мобільних пристроїв?
22. Які комунікаційні технології підтримані в ОС для мобільних пристроїв?
23. Що таке Windows Mobile? Ядро якої ОС використовує Windows Mobile?
24. Які основні офісні і сервісні додатки надає Windows Mobile? Які сучасні версії (види) Windows Mobile використовуються сьогодні?
25. Що таке Symbian OS? Якою компанією і якою мовою розроблена?
26. Які основні можливості Symbian OS?
27. Що таке Google Android? Ядро якої ОС використовує Google Android?
28. Які основні можливості Google Android?
29. Яку реалізацію Java використовує Google Android?
30. Які інструменти розробки додатків для Google Android?

ЧАСТИНА II

РОЗДІЛ 6

ОПЕРАЦІЙНІ СИСТЕМИ LINUX ТА WINDOWS

6.1. Робота з файлами з командного рядка LINUX

Усі імена файлів, каталогів і команд чутливі до великих і малих букв, на відміну від MS-DOS. Наприклад, команда *make* відрізняється від *Make* або *MAKE*. Імена файлів можуть бути довші 8 символів і містити точку або декілька точок на будь-якій позиції.

Отримання довідки за LINUX з командного рядка: *#man man* (Enter). Кожна сторінка довідки – це окремий HTML-файл. Усі вони містяться в каталозі */usr/man*, мають формат *man№.gz*, де *№* – порядковий номер файла, *gz* означає, що файл заархівований командою *gzip*.

Команда *pwd*. Вона виводить робочий каталог і показує, в якому місці файлової системи ви перебуваєте. Наприклад, якщо після команди *# cd /usr/bin* набрати *# pwd*, то на екрані з'явиться */usr/bin*. Для відображення лише поточного робочого каталогу набирають *# pwd -help*. Якщо немає впевненості, в якому місці каталогу ви перебуваєте, запустіть *# / bin/pwd -help*.

Команда *find*. Ця команда шукає файли на диску. Наприклад, для пошуку команди *spell* на диску в каталозі */usr* можна використати таку команду: *# find /usr -name spell -print*. Для пошуку програми в каталозі */usr/bin*, до якої не зверталися останні 100 днів, набирають: *# find /usr/bin -type f -atime +100 -print*. Якщо треба знайти файли в каталозі, які з'явилися за останню добу, набирають: *# find /usr/bin -type f -mtime -1 -print*. Ця команда працює також із шаблонами файлів ** i ?*. Наприклад, для виведення списку всіх файлів з розширенням *.ps* з каталогу */usr*, набирають: *# find /usr -name '*.ps' -print*. Разом з командою *find* застосовують опцію *-xdev*, яка обмежує пошук рамками поточної файлової системи (LINUX), а не Windows і Dos. Наприклад, вам

треба знайти лише у вашому розділі Windows усі файли з розширенням `.sys` у каталозі `/mnt/dos`:

```
#find /mnt/dos -name *.sys -print -xdev.
```

Команда *whereis*. Теж використовується для пошуку файлів. Вказує на тип файла (двійковий чи вихідний) і на розміщення сторінки довідки до цього типу файлів. Наприклад, `# whereis find` (Enter) дає результат: `find: /usr/bin/find /usr/man/man1/find.1.gz`. Тобто відображається інформація про те, що команда `find` є в каталозі `/usr/bin`, а її сторінка довідника – в каталозі `/usr/man/man1`. Якщо шукають лише двійкову версію програми, то набирають: `# whereis -b find` (Enter). Результат: `find:/usr/bin/find`. (показує місцезнаходження лише файла без документації). Якщо файл не знайдено, видається `loo:.` *Whereis* швидша за *find*.

Команда *locate*. Ця команда швидша за *whereis*. Вона не переглядає каталоги, а користується файлом імен усіх файлів. Наприклад: `# locate *.ps`. База даних за об'ємом ~700 000 символів з іменами всіх файлів є в папці `/var/state` й оновлюється командою `# updatedb` (для цього користувач повинен увійти в систему як *root*-оператор).

Команди *whatis* і *apropos*. Ці команди беруть резюме зі сторінки документації за іменем програми. Наприклад, `#apropos search` показує всі програми LINUX, які дозволяють шукати файли за ознакою (зокрема, *find*, *whatis*, *apropos*). Якщо набрати `# whatis who` (Enter), то екран стане чистим і з'явиться рядок: `who (1) – show who is logged on`. Для виходу з *whatis* натискають *Q*. Перед користуванням командою *whatis* у режимі *root*-користувача командою `#makewhatis` оновлюють базу даних резюме. Файл резюме є в `/usr/man/man1`.

Команда *mkdir*. Створює новий каталог. Приклад: `mkdir /home/Peter/test` створює каталог `test` у каталозі `/home/Peter`. Синтаксис: `mkdir <dir1>...<dirN>`, де `<dir1>...<dirN>` – створювані каталоги.

Команда *rmdir*. Вилучає пусті каталоги. Поточний робочий каталог повинен перебувати зовні каталогу, що вилучається. Синтаксис: `rmdir <dir1>...<dirN>`, де

`<dir1>...<dirN>` – каталоги, що вилучаються. Приклад:
`rmdir /home/Peter/papers` вилучає каталог `/home/Peter/papers`, якщо він пустий.

Команда переміщення по дереву каталогів: *cd*. Команда змінює поточний робочий каталог. Після відкриття LINUX ви перебуваєте у домашньому каталозі, ім'я якого складається з `/home/`, за яким слідує ім'я користувача. Наприклад, команда `# cd /usr/bin` переміщує вас у каталог `/usr/bin`. Щоб потрапити в каталог `/usr`, треба набрати `# cd ..`, а для потрапляння в кореневий каталог – `# cd ../../`. Повернутися в домашній каталог можливо командою `# cd` без аргументів або `# cd ~`. Символ «`..`» посилається на поточний каталог, «`...`» – на попередній у дереві каталог. Наприклад, `#cd ../foo` переводить з поточного каталогу в `../foo`.

Команда перегляду вмісту каталогів: *ls*. Видає інформацію про файли в каталозі.

Синтаксис: `ls <file1>...<fileN>`, де `<file1>...<fileN>` – імена файлів або каталогів, інформацію про які треба показати на екрані.

Опції. Найчастіше використовується: `-F` (для представлення інформації про типи файлів) та `-l` (видає в довгому (*long*) форматі інформацію про розміри файлів, їх власників, права доступу тощо). Приклад: `ls -lF /home/Peter` видасть вміст каталогу `/home/Peter`. Риска зліва від назви означає, що ця назва – каталог. Якщо такої риски немає, то це – файл.

Команда *rm*. Вилучає файли. У LINUX вилучені файли відновити неможливо. Синтаксис: `rm <file1>...<fileN>`, де `<file1>...<fileN>` – імена файлів для вилучення. Опція `-i` вимагає вашої згоди на вилучення файла. Приклад: `rm -i /home/Peter/joe /home/Peter/frog` вилучає файли `joe` та `frog` в каталозі `/home/Peter`.

Команда *cp*. Копіює файл(и) у файл або каталог. Синтаксис: `cp <file1>...<fileN> <destination>`, де `<file1>...<fileN>` – імена файлів, що копіюються, а `<destination>` – файл або каталог, в який копіюють. Приклад:

`cp ../frog joe` копіює файл `../frog` у файл або каталог `joe`. Декілька файлів однією командою `cp` можна скопіювати лише в каталог; у файл можна скопіювати лише один файл. Для посилання на поточний каталог використовують «.». Наприклад:

```
/home/Peter/loo# cp /etc/termcap .  
/home/Peter/loo# cp /etc/shells .  
/home/Peter/loo# ls -F  
shells  termcap
```

Команда *mv*. Переміщує файл(и) в інший файл або каталог. Ця команда нееквівалентна копіюванню з наступним знищенням оригіналу. Застосовується і для перейменування файлів, як команда `RENAME` з `MS-DOS`. Синтаксис: `mv <file1>... <fileN> <destination>`, де `<file1>...<fileN>` – імена переміщуваних файлів, а `<destination>` – ім'я файла або каталогу, в який переміщують.

Приклад 1: `mv ../frog joe` переміщує файл `../frog` у файл або каталог `joe`.

```
Приклад 2: /home/Peter/loo# mv termcap sells  
/home/Peter/loo# ls -F  
bells  sells  shells
```

Видно, що *termcap* більше не існує, а на його місці файл `sells`.

Зауваження. Команди `mv` і `cp` знищують вміст файла, в який вони пишуть (якщо він є на диску) за замовчуванням. Тому у разі перенесення файла в інший каталог перевірте, чи існує в ньому файл з таким самим ім'ям.

Команда *more*. Видає вміст названих файлів поєкранно.

Синтаксис: `more <file1> ... <fileN>`, де `<file1> ... <fileN>` – відображувані файли. Приклад: `more papers/final` представляє файл `papers/final`.

Команда *cat*. Використовується для конкатенації і для видачі повного вмісту файла за один раз. Синтаксис: `cat <file1> ... <fileN>`, де `<file1> ... <fileN>` – файли до видачі.

Приклад: `cat letters/mdw` видає на дисплей файл `letters/mdw`.

Команда *grep*. Видає всі рядки у вказаних файлах, які містять заданий зразок.

Синтаксис: `grep <pattern> <file1> ... <fileN>`, де `<pattern>` – зразок і `<file1> ... <fileN>` – файли, в яких відбувається пошук. Приклад: `grep loom /etc/hosts` видасть усі рядки, в яких файл `/etc/hosts` містить зразок «loom».

Права доступу та їх зміна

Команда `ls` з опцією `-l` видає на екран список файлів певного каталогу в «довгому» форматі з інформацією про права доступу:

```
/home/Peter/loo# ls -l stuff
-rw-r--r--  1 Peter  users      505 Mar 13
19:05 stuff
```

Перше поле в цьому рядку – це права доступу (за порядком вказано права власника, групи і всіх інших). `Peter` – власник файла, `users` – група, `stuff` – ім'я файла. Символ «-» – це тип файла (звичайний файл (не каталог і неспеціальний файл будь-якого пристрою)). Наступні 3 символи «rw-» – це права доступу `Petera`. Символ «r» означає «*read*» (читати), «w» – «*write*» (писати). Отже, `Peter` може читати файл `stuff` і писати в нього, але виконувати цей файл йому заборонено (право виконання надається після «rw» третім символом «x», тобто «*execute*» (виконати)). Наступні 3 символи `r-` є правами доступу групи `users` для цього файла (лише читання). Усім іншим дозволено те саме («r--» – лише читання).

Приклад 1: `-rwxr-xr-x` (власник файла може читати, писати і виконувати файл. Члени групи і всі інші користувачі можуть читати і виконувати файл).

Приклад 2: `-rw-----` (власник файла може читати і писати у файл. Усім іншим доступ до файла закрито).

Приклад 3: -rwxrwxrwx (усі можуть читати, писати і виконувати файл).

Права доступу змінюються командою з таким синтаксисом: `chmod {a,u,g,o}{+,-}{r,w,x} <filenames>`, де параметр *a* означає всі, *u* – користувач, *g* – група, *o* – інші, + означає надати права, – означає забрати права. Наприклад: **chmod a+r stuff** надає всім користувачам право читати файл *stuff*; **chmod og-x stuff** – забирає право на виконання у всіх, крім власника; **chmod u+rwx stuff** – дозволяє власнику всі дії з файлами; **chmod o-rwx stuff** – забороняє всі дії з файлами користувачам категорії *other*.

6.2. Мова програмування Perl у Linux для Perl-скриптів

ОС Linux має вбудовану мову програмування Perl (Practical Extraction and Report Language). Perl безкоштовна, структурована, різностороння і гнучка мова програмування, якою написано більшість скриптів для Web. Perl інтерпретована, тобто Perl-програми не треба компілювати. Perl має широкий спектр інструментів для роботи з файлами.

На Perl знак # (дієз) означає коментар (ігнорується текст після дієзу до кінця рядка).

Перший рядок (`#!/c:/usr/bin/perl` або `#!/c:/usr/local/bin/perl`) будь-якого Perl-скрипта дає команду Linux запуску скрипта інтерпретатором Perl, інсталюваним за адресою `c:/usr/bin/perl` чи `c:/usr/local/bin/perl`. Цей перший рядок є коментарем, але без нього скрипт може не спрацювати.

Застосування PERL-дебагера. У ньому легко за рядками перевіряти роботу скриптів. Perl-дебагер вбудовано в Perl і запускається з командного рядка так: `C:\PERL> Perl -d hello.pl <ENTER>`. Perl завантажить скрипт *hello.pl* і почне відлагодження. У Linux також можна написати рядок `#!/usr/bin/perl -d` на самому початку Perl-скрипта. Загрузка Perl-дебагера без загрузки скрипта: `C: \> Perl -de 0 <ENTER>` (ключ *-d* вказує Perl запустити

дебагер, а аргумент *-e 0* – виконати Perl-скрипт з 0 рядків). За нормальної роботи Perl-дебагера на екран виводиться: *Loading DB routines from \$RCSfile: perl5db.pl,v \$\$Revision: 4.0.1.3 \$\$Date: 04/09/09 13:43:57 \$ Emacs support available. Enter h for help. main '(p1000159:1): DB<1>*. Якщо виводиться «*Can't locate perl5db.pl @INC*», то Perl треба перевстановити. Команди дебагера: *- h* – виведення списку команд дебагера, *- n* – виконувати до наступного виразу; *- <CR>* – повторити останню команду *n* або *s*, *- p* вираз – друкувати вираз, *- q* – закінчити роботу, *- r* – виконувати до виходу з підпрограми, *- s* – крок у скрипті (із входом у підпрограму).

Приклади виведення на екран сповіщення *Hello Friends*:

```
1) DB<1> printf "Hello Friends\n"; <ENTER>
Hello Friends DB<2> ;
```

```
2) <ENTER> DB<2> p "Hello Friends\n"
<ENTER> Hello Friends DB<3>.
```

Для введення декількох рядків у дебагер у кінці кожного рядка вставляють символ продовження рядка *</>*: *DB<3> for (\$i = 0; \$i < 10; \$i++){\<ENTER>cont: print \$i;\<ENTER> cont: } <ENTER> 0123456789 DB<4>*.

Скалярні змінні. Тип числових або символічних даних вказують символом *\$*: *\$ім'я_скаляра = 4*; (скалярній змінній *\$ім'я_скаляра* присвоєно число 4). Скалярні змінні можна додавати і віднімати.

Список – це впорядковані скаляри, записані через кому в круглих дужках: *(5, \$x, 12, 2+\$y)*.

Масив – це змінна, яка містить список. Перед змінною-масивом ставиться знак *@*:

@ім'я_масиву = (5, \$x, 12, 2+\$y);

Доступ до елемента масиву отримують так: *\$ім'я_масиву [номер_скаляра]*; Номер_скаляра відраховується з нуля. Отже, звернення до змінної *\$x* в цьому масиві таке: *\$ім'я_масиву [1]*.

Хеш – це асоціативний масив. Кожний елемент хеша має свій «ключ» доступності. Назва хешу починається із символу *%*. Додають новий елемент у хеш так: *\$хеш {"ключ"} = "значення"*.

Тепер %хеш містить «значення». Видобувають елемент «значення» так: `$хеш {"ключ"}`.

Керуючі структури. Далі скорочення «оп» означає «оператор». Нижче наведено синтаксис керуючих Perl-структур:

Керуюча структура if/else:

```
if (умова_1) {
    оп_1; оп_2; ... # ці оператори
    виконуються,   # якщо умова_1
    правильна
} else {

    оп_3; оп_4; ... # ці оператори вико-
    нуються,
    #якщо умова_1 хибна
}
```

Керуюча структура for:

```
for (вираз_1; умова_1; вираз_2) {
    оп_1; оп_2; ...
}
# вираз_1 виконується один раз
на початку циклу;
# блок операторів виконується,
доки #умова_1 правильна;
# вираз_2 виконується в кінці
кожного #циклу
```

Керуюча структура while:

```
while (вираз_1) {
    оп_1; оп_2; ...
}
# блок операторів виконується,
# доки вираз_1 правильний
```

Керуюча структура until:

```
until (вираз_1) {
    оп_1; оп_2;
}
# блок операторів виконується,
# доки вираз_1 хибний
```

Керуюча структура do/while:

```
do {оп_1; оп_2;
} while вираз_1;
# блок операторів виконується
один раз.
# Далі блок виконується,
доки вираз_1 правильний
```

Керуюча структура do/until:

```
do {оп_1;оп_2;
} until вираз_1;
# блок операторів виконується
один раз.
# Далі блок виконується,
доки вираз_1 хибний
```

Керуюча структура foreach:

```
foreach $x (@list) {оп_1; оп_2; ...}
# скалярній змінній $x за чергою
присвоюються #значення зі списку
@list.
# З кожним наступним значенням $x
виконується #блок операторів
```

Функції користувача. Свою функцію пишуть у кінці скрипта. Її синтаксис такий:

sub ім'я_функції {on_1; on_2; ...} . Викликається вона вставкою у скрипт рядка: *ім'я_функції()*;

Зарезервовані змінні

Для використання в Perl довгих імен змінних у заголовку скрипта пишуть *use English*;

У Perl є змінні лише для читання (значення в них не записати не можна):

\$_ – в неї за умовчанням можна вводити, присвоювати, заносити результат пошуку за заданим зразком: *while(\$_ = <>){...}* ідентичне за дією *while(<>){...}*;

\$<digit> – доступна лише для читання, як і змінні **\$&**, **\$'**, **\$' i \$+**;

\$. – містить номер останнього прочитаного рядка з останнього прочитаного файлу. Доступна лише для читання;

\$/ – містить символ розділення записів, що вводяться. За замовчуванням містить символ переведення рядка;

\$| – містить 0 за умовчанням. Якщо в ній не 0, очищаються буфери щоразу після виведення на друк, екран тощо;

\$_, – містить символ-розділювач полів для оператора друку;

\$ – містить символ-розділювач записів для оператора друку. Задають **\$** замість набирання *n* у кінці друку;

\$" – подібна **\$**, використовується під час звернення до списку величин у подвійних лапках. За замовчуванням містить символ-пробіл;

\$; – містить символ-розділювач для емуляції багатовимірних хешів. У разі звернення до елемента хеша як *\$toy{\$a,\$b,\$c}* реально виконається *\$toy{join(\$;,\$a,\$b,\$c)}*. Не плутати з *@toy{\$a,\$b,\$c}*, що таке саме за дією, як і *(\$toy{\$a},\$toy{\$b},\$toy{\$c})*. За замовчуванням містить значення `\034`;

\$# – формат для друку чисел (містить початкове значення `%.20g`);

\$% – містить номер поточної сторінки, що виводиться;

\$= – містить кількість рядків для друку поточної сторінки (зазвичай містить значення 60);

\$- – містить кількість ще не надрукованих рядків сторінки для каналу виведення під час друкування;

\$~ – містить ім'я поточного формату сповіщень (зазвичай ім'я дескриптора файла);

\$^ – містить ім'я поточного формату заголовку сторінки (зазвичай ім'я дескриптора файла з `_TOP` в кінці);

\$: – містить символи, після яких виведення рядка починається з нового рядка;

\$! – якщо це змінна-число, то містить поточне значення *errno* (номер помилки). Якщо це змінна-рядок, то містить відповідне системне сповіщення про помилку;

\$@ – містить сповіщення про синтаксичну помилку під час виконання останньої команди *eval()*. Сповіщення в ній не накопичуються. У разі виконання *eval()* без помилок змінна містить 0;

\$\$ – містить ідентифікатор поточного процесу;

\$< – містить ідентифікатор користувача (UID), якому належить поточний процес;

\$> – містить ефективний UID поточного процесу;

\$() – містить ідентифікатор групи (GID) користувача, якому належить поточний процес;

\$) – містить ефективний GID поточного процесу;

\$0 – містить ім'я файлу скрипта;

\$ARGV – містить ім'я поточного файлу, з якого відбувається читання;

@ARGV – містить масив аргументів командного рядка, які були передані скрипту.

Наприклад, з Perl-скрипта можна отримати доступ до аргументів командного рядка шляхом запису в Perl-скрипті аргументів командного рядка у спискову змінну *@ARGV*. Для виведення аргументів командного рядка на екран: *while (\$arg = shift @ARGV) { print "\$arg\n"; }*;

@INC – містить список точок входу у скрипт з конструкціями *do EXPR*, *require* та *use*;

%INC – містить входи для кожного файла, який включається за допомогою операторів *do* або *require*. Ключами є імена файлів, а значеннями – місця їх розміщення;

%ENV – містить поточне оточення процесу. Змінною вмісту хеша можна змінити оточення породженого (дочірнього) процесу;

%SIG – цей хеш використовується для встановлення обробників різних сигналів.

Дескриптори файлів

Доступ до файла дає дескриптор – символічне ім'я в Perl-скрипті для посилання на файл, пристрій, сокет, програмний канал (дескриптор програмного каналу пов'язує процеси). Ім'я дескриптора файла пишеться для зручності пошуку в скрипті лише великими літерами і не може бути ім'ям зарезервованої Perl-функції. Одному дескриптору може відповідати лише один файл, хоча у різних місцях програми це може бути щоразу інший файл.

Дескриптор файла в Perl і в Linux – це два різних дескриптори. Дескриптор файла в Linux – це 32-бітна адреса системної області для посилань на відкритий файл. Ця область містить інформацію про цей файл. Дескриптор файла в Perl – це не адреса, а ім'я, вигадане програмістом для посилань на файл. Отримати дескриптор файла в Linux з Perl-програми можна командою *fileno()*.

Створює дескриптор файла функція *open()* з двома параметрами – ім'ям дескриптора та ім'ям файла з вказаним режимом доступу: *open(DESCRYPTOR, "> /temp/myfile.txt");* *#open* створює дескриптор *DESCRYPTOR*, приєднує його до файла */temp/myfile.txt* і відкриває файл для запису (на це вказує символ ">"). Приклад запису в цей файл значення змінної *var*: *print DESCRYPTOR \$var;* Дескриптор не можна присвоїти, зберегти в змінній або передати як параметр у функцію, оскільки дескриптор – не змінна (його ім'я не містить префікса змінної Perl (\$, @

або %)). Для цих дій перед ім'ям дескриптора пишуть префікс * – посилання на глобальний тип даних. Наприклад, записати у файл значення змінної можна ще й так: *\$name1 = *DESCRIPTOR; print \$name1 \$var*; #змінна *\$name1* заміщує *DESCRIPTOR*, в який виводиться значення змінної *\$var*.

У будь-якому Perl-скрипті завжди є три наперед заданих дескриптори (*STDIN*, *STDOUT* і *STDERR*), де *STDIN* пов'язаний з клавіатурою, *STDOUT* і *STDERR* – з екраном монітора. *STDIN* використовує функцію *open*, якщо в командному рядку виклику Perl-скрипта немає списку файлів. *STDOUT* за замовчуванням використовують функції *print* і *die*, а *STDERR* – функцію *warn*. Можна перенаправляти стандартне введення і виведення в інші файли дописуванням префіксів '>' (у файл) і '<' (з файла): *perl script1.pl <in.dat >out.dat*; у разі виконання *script1.pl* дані з файла *in.dat* запишуться у файл *out.dat* без показу на екрані. Перенаправлення з Perl-скрипта можливе у виді: *open(STDIN, "in.dat"); open(STDOUT, ">out.dat"); open(STDERR, ">err.dat")*. Тепер усе стандартне введення / виведення йтиме крізь вказані в *open()* файли. Таке перенаправлення можна робити в Perl-скрипті лише одноразово (функція *open()* не повертає початкові призначення дескрипторів *STDIN*, *STDOUT* і *STDERR*).

Вибрані вбудовані Perl-функції для роботи з файлами і каталогами

binmode FILEDESCR – для читання або запису у файл за його дескриптором у бінарному режимі;

caller EXPR – видає контекст поточного виклику підпрограми. Якщо *EXPR* скаляр, то видає *TRUE* (у разі перебування функції в тілі підпрограми), *eval()* або *require()*, інакше – *FALSE*. Якщо *EXPR* список, то видає: *(\$package, \$filename, \$line) = caller*. З аргументом *EXPR* видає інформацію для використання дебагером під час друкування карти стеку. Значення *EXPR* відмічає глибину стека до поточного запису: *(\$package, \$filename, \$line, \$subroutine, \$hasargs, \$wantargs) = caller(\$i)*;

chdir EXPR – змінює поточну директорію на вказану в *EXPR*, якщо остання існує. Без *EXPR* поточною стає домашня директорія. Видає *TRUE* в разі успіху, інакше – *FALSE*;

chmod LIST – змінює права доступу до файлів зі списку *LIST*. Першим аргументом списку є цифрова (зазвичай вісімкова) маска доступу. Видає кількість файлів з успішно зміненими правами доступу. Наприклад: *\$cnt = chmod 0700 'loo', 'tar'; chmod 700 @executables; chmod 0666 'f1', 'f2', 'f3'*;

chown LIST – змінює власника або групу, якій належить список файлів. Першими двома аргументами завжди є *uid* та *gid*. Видає кількість успішних змін;

close FILEDESCR – закриває файл з дескриптором *FILEDESCR*. Наприклад:

```
open(FD5 'usr/home/stud'); ... close FD5;
```

closedir DIRDESCR – закриває каталог, відкритий функцією *opendir*;

dbmopen ASSOC, DBNAME, MODE – пов'язує *dbm(3)* або *ndbm(3)* файл з асоціативним масивом. *ASSOC* – ім'я асоціативного масиву. *DBNAME* – ім'я бази даних (без *.dir* або *.pag* розширення). Якщо база даних не існує, вона створюється з правами доступу, вказаними в *MODE*:

```
dbmopen(%HIST, '/usr/lib/news/history', 0600); while (($key, $val) =  
each %HIST){ print $key, '=', unpack('L', $val), "\n";}  
dbmclose(%HIST);
```

dbmclose ASSOC – розриває зв'язок між файлом і асоціативним масивом;

die LIST – за межами *eval()* друкує значення *LIST* у *STDERR* і виходить з програми з поточним значенням *\$!*. Якщо в *\$!* нуль, то приймає значення *\$? >> 8*, а якщо значення *\$? >> 8* нульове, то 255. Всередині *eval()* сповіщення про помилку заноситься в змінну *\$@* і *eval()* переривається з невизначеним значенням. Приклад: *open(FL, "/root/rm-rf") || die "Can't open file.\n";*

do EXPR – вважає *EXPR* ім'ям файла і запускає вміст цього файла як програму на *Perl*. Часто нею включають у скрипт бібліотечні підпрограми. Наприклад: *do 'text.pl'*; ідентичне:

eval 'cat text.pl'; Підключати бібліотечні модулі зручніше функціями *use* та *require*;

eof FILEDESCR – видає 1, якщо наступне зчитування видає кінець файла або якщо *FILEDESCR* не було відкрито. Без аргументу *eof* опрацьовує останній файл, що зчитувався. Функція мало використовується, оскільки в *Perl* оператори читання видають невизначене значення в кінці файла;

getc FILEDESCR – читає і видає наступний символ з файла читання, приєднаного до *FILEDESCR* або пустий рядок у випадку кінця файла. Без *FILEDESCR* зчитує з *STDIN*.

Код нижче читає і виводить 15 байт з файла *"file.txt"*: *open (file, "file.txt");*

```
while ($u<=15){ print getc(file);} continue{$u++} close file;
```

link OLDFILE, NEWFILE – створює файл *NEWFILE*, приєднаний до файла *OLDFILE*. В *UNIX* для одного файла можна створити декілька імен. Видає 1 в разі успіху і 0, якщо інакше;

mkdir FILENAME, MODE – створює директорію з іменем *FILENAME* і правами доступу, вказаними в змінній *MODE* (вісімкове число 0777). У разі успіху видає 1, інакше видає 0 і встановлює значення змінної *\$(errno)*. Якщо задано ім'я каталогу без шляху до нього, то він створюється в поточному каталозі;

open FILEDESCR, EXPR – відкриває файл з іменем *EXPR* і ставить йому відповідну файлову змінну *FILEDESCR* (дескриптор файла). Без *EXPR* змінна з іменем *FILEDESCR* містить ім'я файла. Спецсимвол (<,>,>>,<>,<+>,>+>) перед ім'ям файла визначає режим доступу до файла щодо його відкриття. Наприклад: *open (FILEDESCR, "EXPR");* або *open (FILEDESCR, "<EXPR");* – відкриває файл тільки для читання (файл має існувати);

open (FILEDESCR, ">EXPR"); – відкриває файл для запису в його початок зі знищення попереднього вмісту файла (якщо файл не існує, то створюється);

open (FILEDESCR, ">>EXPR"); – відкриває файл для дозапису в його кінець без знищення попереднього вмісту файла (якщо файл не існує, то створюється);

open (FILEDESCR, "+<EXPR"); – відкриває файл для читання і запису без знищення попереднього вмісту файла;

open (FILEDESCR, "+>EXPR"); – створює файл для читання і запису зі знищенням попереднього вмісту файла;

open (FILEDESCR, "+>>EXPR"); – створює файл для читання і дозапису в кінець без знищення попереднього вмісту файла. Після закінчення роботи з файлом завжди пишуть *close (FILEDESCR);* (закривається файл *FILEDESCR*).

Спецсимволом також може бути знак «|», який може ставитися перед ім'ям або відразу ж після імені файла. Знак «|» перед ім'ям файла означає, що воно є командою створення програмного каналу – вихід з *FILEDESCR* передається на вхід *EXPR*. Наприклад: *open (FILEDESCR, "|EXPR");* – направити інформацію на вхід програми. Знак «|» після імені файла означає, що вихід з *EXPR* передається на вхід *FILEDESCR*, тобто з *FILEDESCR* відбуватиметься читання. Наприклад: *open (FILEDESCR, "EXPR|");* – зчитати інформацію з виходу програми. Фрагмент програми показує використання функцій *open* і *close*:

```
open(InFile, " proba.dat") || die; # від-
криваємо для читання proba.dat
open(OutFile, ">proba.dat") || die;    #
створюємо proba.dat
$AuxFile      =      ">>proba.dat";    open(Aux,
$AuxFile) || die; # відкриваємо для      дозапису
proba.dat      close(InFile);      close(OutFile);
close(Aux);
```

open DIRDESCR, EXPR – відкриває директорію з ім'ям *EXPR*, видає *TRUE* в разі успіху;

opendir DIRDESCR, DIRNAME – відкриває директорію з ім'ям *DIRNAME*. Приклад Perl-скрипта, який перевіряє, чи є всі файли каталогу двійковими (без перевірки вмісту вкладених каталогів):

```
#!/ perl -w opendir FDIR, "/usr/prog"; while
($name=readdir FDIR) {next if -d $name; # каталог
print("/usr/prog/$name: двійковий\n") if -b $name; #Двійковий файл
} closedir FDIR;
```

print FILEDESCR, LIST – друкує рядок або декілька рядків, розділених комою. *FILEDESCR* може бути ім'ям скалярної змінної, яка містить дескриптор файла. Без цієї змінної друк йде у вибраний канал виводу. Без змінної *LIST* друкує змінну *\$_* в *STDOUT*;

printf FILEDESCR, LIST – еквівалент `print FILEDESCR, sprintf (LIST)`. Перший аргумент *LIST* інтерпретується як формат друку;

read FILEDESCR, SCALAR, LENGTH, OFFSET – зчитує *LENGTH* байт даних з *FILEDESCR* у змінну *SCALAR*. Видає кількість зчитаних байт або невизначеність у разі помилки. Для зчитування даних не з початку рядка вводять значення змінної *OFFSET*.

Положення вказівника зберігається. Код нижче читає і виводить 15 байт з файла "file.txt": `open (file,"file.txt"); read("file",$u,5); print $u,"<BR>"; read("file",$u,10); print $u; close file;`

readdir DIRDESCR, DIRNAME – для відкритого каталогу видає список імен усіх файлів каталогу. У скрипті допускаються дескриптори файла і каталогу з однаковими іменами: `open FF, "/usr/out.dat" # дескриптор файла opendir FF, Rusr" # дескриптор каталогу`. Приклад використання функції *readdir* для виведення на екран списку файлів з поточного каталогу: `opendir(Dir, $INC[2]) || die; while ($file = readdir(Dir)) { print "$file \n" } closedir(Dir);`, де змінна *\$INC[2]* дає доступ до поточного каталогу. Змінюючи *\$INC[2]* на *\$ARGV[0]*, скрипт виводить на екран список файлів з каталогу, вказаному в командному рядку;

readlink EXPR – видає значення символьного посилання, якщо посилання існує. Якщо посилання не існує, то видає *fatal error* і встановлює значення змінної *\$!*. За замовчуванням опрацьовує змінну *\$_*;

require EXPR – підключає модулі. Приклад: `require "oraperl.pm";`

rename (старе_ім'я, нове_ім'я_файла); – перейменовує файл (видає 1 у разі успіху, інакше – 0);

rewinddir DIRDESCR – поточну позицію в каталозі встановлює на початок (можна повторно читати імена файлів каталогу, не закриваючи його). Єдиним параметром цієї функції є дескриптор відкритого каталогу;

rm FILENAME – вилучає файл або директорію з заданим ім'ям. Видає 1 у разі успіху, інакше – 0 і встановлює значення змінної *\$!*. За замовчуванням опрацьовує аргумент *\$_*;

rmdir DIRNAME – вилучає пустий каталог (без вкладених пустих каталогів). Якщо параметра не задано, то вставляють спеціальну змінну *\$_*, яка містить значення «істина» у разі успішного знищення каталогу і пояснення помилки за неможливості знищення;

seek FILEDESCR, POSITION, WHENCE – встановлює курсор у файлі, визначеному в змінній *FILEDESCR*, на позицію *POSITION* у режимі, вказаному в змінній *WHENCE* (або відлік). Якщо в змінній *WHENCE* 0, то позиція починає відлік від початку файла, якщо 1 – від поточної позиції і якщо 2, то від кінця файла. Видає 1 у разі успіху і 0 – інакше;

select FILEDESCR – видає поточний вибраний *FILEDESCR*. Направляє виведення у *FILEDESCR*;

stat FILENAME – видає 13 – елементний масив параметрів вказаного файла або пустий список у разі помилки. Наприклад: *(\$dev,\$ino,\$mode,\$nlink,\$uid,\$gid,\$rdev,\$size,\$atime,\$mtime,\$ctime,\$blksize,\$blocks) = stat(\$filename);* *\$dev* – ім'я пристрою; *\$ino* – номер і-вузла; *\$mode* – права доступу; *\$nlink* – кількість зв'язків; *\$uid* – ідентифікатор власника; *\$gid* – ідентифікатор групи; *\$rdev* – тип пристрою; *\$size* – розмір файла в байтах; *\$atime* – дата останнього звернення; *\$mtime* – дата останньої модифікації; *\$ctime* – дата останньої зміни статусу; *\$blksize* – розмір блока на диску; *\$blocks* – кількість блоків у файлі. Код виведення розміру файла в байтах: *\$u = (stat("file.txt"))[7];print \$u;*;

sysopen ДЕСКРИПТОР, ІМ'Я, ПРАПОР [, ДОЗВІЛ]; – може, крім завдання режиму відкриття файла, задавати права доступу до файла: режим відкриття файла задає ПРАПОР – число

як результат побітового *АБО* (`|`) над константами режимів з модуля `Fcntl`. Константи режиму доступу до файла: `O_RDONLY` (тільки читання), `O_WRONLY` (тільки запис), `O_RDWR` (читання і запис), `O_CREAT` (створення файла, якщо він не існує), `O_EXCL` (завершення з помилкою, якщо файл уже створено), `O_APPEND` (додавання в кінець файла `%`). Права доступу (параметр ДОЗВІЛ) задаються у вісімковій системі з урахуванням поточного значення маски доступу до процесу з функції `umask()`. Якщо цей параметр не задано, то Perl використовує значення `0666` для звичайних файлів. Для каталогів і виконуваних файлів використовують `0777`. Приклад операцій відкриття файлів функцією `open()` та `sysopen()`:

```
use Fcntl; open FF, "< file.txt"; sysopen
FF, "file.txt", O_RDONLY; # лише читання open
FF, "> file.txt"; sysopen FF, "file.txt",
O_WRONLY | O_CREAT | O_TRUNC; # лише запис
(створює новий файл або очищує існуючий) open
FF, ">> file.txt"; sysopen FF, "file.txt",
O_WRONLY | O_CREAT | O_APPEND; # дозапис в кі-
нець (створюється, якщо не існує) open FF, "+<
file.txt"; sysopen FF, "file.txt", O_RDWR; #
читання/запис існуючого файла open FF, "+>
file.txt"; sysopen FF, "file.txt", O_RDWR |
O_CREAT | O_TRUNC; # читання/запис (файл очища-
ється);
```

tell FILEDESCR – видає поточну позицію курсору у відкритому файлі `FILEDESCR`. Без аргументу опрацьовує файл, який читався останнім. Наприклад, наступний код читає 15 байт з файлу "file.txt" (довжина файлу повинна бути більше 15 байт):

```
print "<pre>"; open (file, "file.txt");
while (tell(file) < 15){print getc(file);}
print "</pre>";
```

truncate FILEDESCR, LENGTH – обрізає файл `FILEDESCR` до заданої довжини.

Приклад запису у файл `file.txt` рядка "This is a sample file" і вкорочення його до 20 байт:

```
#!/usr/bin/perl print "Content-type: text/html\n\n"; $string = "This is a sample file"; print "Write to file: $string"; open (file,">file.txt"); print file $string; close file; truncate ("file.txt",20); open (file, "file.txt"); $string=<file>; close file; print "Read from file: ", $string;
```

unlink LIST – вилучає список файлів і видає кількість успішно вилучених файлів. Не вилучає каталоги, якщо ви не *root*-користувач. Безпечніше використати функцію *rmdir*;

use Module LIST – приєднує модуль до програми: *use strict qw(subs,vars,refs);*

utime Date1, Date2, список_файлів – змінює дату звернення і модифікації файлів зі списку. Перші два елементи списку повинні вказувати нове значення дати звернення і модифікації. Видає кількість змінених файлів;

write – створює запис, який може складатися з кількох рядків у відповідному файлі, з використанням формату, що відповідає цьому файлу. Формат для поточного каналу виведення встановлюється присвоєнням змінній `$~{ }` назви формату;

-X – перевірка файла на можливість дій з ним. Це унарний оператор з одним аргументом – або ім'ям файла, або вказівником файла. Перевіряє одну з умов. Без аргументу береться значення змінної `$_`. Аргумент можна писати в круглих дужках. 'X' має такі значення:

-r – файл дозволено читати ефективним *uid/gid*; **-w** – у файл дозволено записувати ефективним *uid/gid*; **-x** – файл дозволено виконувати ефективним *uid/gid*; **-o** – файл належить ефективному *uid* (ідентифікатор користувача); **-R** – файл дозволено читати реальним *uid/gid*; **-W** – у файл дозволено записувати реальним *uid/gid*; **-X** – файл дозволено виконувати реальним *uid/gid*; **-O** – файл належить реальному *uid*; **-e** – файл існує; **-z** – файл пустий; **-s** – файл непустий; **-f** – звичайний текст; **-d** – директорія; **-l** – символічне посилання; **-p** – pipes (конвеєр); **-S** – socket (сокет); **-b** –

спеціальний блочний пристрій; *-c* – спеціальний символний пристрій; *-t* – вказівник на пристрій *tty*; *-u* – встановлено біт *setuid*; *-g* – встановлено біт *setgid*; *-k* – встановлено біт *sticky*; *-T* – текстовий файл; *-B* – двійковий файл; *-M* – «вік» файла в днях на момент старту скрипта; *-A* – днів з останнього читання; *-C* – днів з останньої модифікації *inode*.

Приклад перевірки наявності файла "file.txt" у поточній директорії і виведення за наявності кількості днів з моменту останнього звернення до нього:

```
if (-e("file.txt")){print (-A("file.txt"))}
else {print "Файл не створено"}.
```

Синтаксис регулярних виразів для обробки тексту

Регулярний вираз – це зразок, що складається із символів. Він використовується для аналізу скриптом вхідних даних з урахуванням пробілів, ком, символів табуляції та інших розділювачів. У регулярних виразах Perl для скороченого запису використовує символи:

Символ	Опис
.	Відповідає будь-якому символу (крім символу нового рядка)
(..)	Групує послідовність елементів
+	Задовольняє попередньому зразку один або більшу кількість разів
-	Задовольняє зразку нуль або один раз
*	Відповідає зразку один або нуль разів
[...]	Відповідає символу із заданої множини
[^...]	Відповідає символу з множини, отриманої запереченням
(...)	Відповідає одній з альтернатив
^	Відповідає початку рядка
\$	Відповідає зразку в кінці рядка
{n,m}	Відповідає зразку від n до m разів

{n}	Відповідає зразку точно n разів
{n,}	Відповідає зразку мінімум n разів
\n\t etc.	Відповідає знаку нової лінії, символу табуляції тощо
\b	Відповідає на межі слова
\B	Відповідає всередині меж слова
\d	Відповідає цифрі
\D	Відповідає не цифрі
\s	Відповідає пробілу
\S	Відповідає не пробілу
\w	Відповідає букві або цифрі
\W	Відповідає символу, що не є ні буквою, ні цифрою

У Perl регулярні вирази (зразки, шаблони) обмежуються з обох боків слешами (похилими рисками), наприклад у вигляді `/pattern/`.

Наприклад:

```
# наступні регулярні вирази істинні, якщо:
/ig/      # рядок містить 'ig'
/(b|d|f)ig/      # рядок містить 'big',
'dig' або 'fig'
/[0-9]+/      # рядок містить число
/[A-Za-z][A-Za-a0-9_]* / # рядок містить
ідентифікатор
```

Детальніше з роботою з файлами в Perl можна ознайомитися на сайтах: <http://www.PERL.org.ru> або <http://www.perl.com>

Приклад. Створити Perl-скрипт, який виводить на екран дату-час, інформацію про вільне місце на дисках та про використання оперативної пам'яті комп'ютера.

Рішення. Створюємо в *Kword* або *StarOffice* такий текст програми і зберігаємо його з розширенням `.txt` або `.pl`:

```
use Term::ANSIColor; system clear; print
color("magenta"); system date;
ptint color("white"), "\n"; system df;
print color ("yellow"), "\nOperative memory\n";
system free; print color("reset"), "\n";
```

Під час збереження програми даємо ім'я файлу: *fds.txt*. Запускається Perl-скрипт на виконання з командного рядка Linux так: *[stud@localhost Perl] \$ perl -w fds.txt*.

6.3. Створення, відслідковування і знищення процесів в ОС Linux

Команда ps. Назва: *ps* – видача інформації про стан процесів.

Синтаксис: *ps [-i] [-d] [-a] [-f] [-l] [-n* файл із системою] *[-t* список терміналів] *[-p* список ідентифікаторів процесів]; *[-u* список ідентифікаторів користувачів] *[-g* список ідентифікаторів лідерів груп].

Опис. Команда *ps* видає інформацію про активні процеси. За замовчуванням інформація дається тільки про процеси, що асоціюються з цим терміналом. Виводяться ідентифікатор процесу, ідентифікатор терміналу, витрачений до цього моменту час ЦП та ім'я команди. Якщо потрібна інша інформація, варто користуватися опціями. Деякі опції мають один аргумент або список аргументів. Аргументи в списку можуть бути або відокремлені один від одного комами, або всі разом поміщені в подвійні лапки і відокремлені пропусками чи комами.

Аргументи в списку процесів і в списку груп мають бути числами.

Командою *ps* обробляються такі опції: *-i* – вивести інформацію про усі процеси; *-d* – вивести інформацію про усі процеси, крім лідерів груп; *-a* – вивести інформацію про всі процеси, що найчастіше запрошуються, тобто про усі процеси, крім лідерів груп, і процеси, що не асоціюються з терміналом; *-f* – генерувати повний лістинг; *-l* – генерувати лістинг у довгому форматі; *-n* – файл із системою (вважати, що операційна система завантажена з файла із системою, а не з файла /unix); *-t* – список терміналів; видавати інформацію тільки про процеси, що асоціюються з терміналами із заданого списку терміналів.

Термінал – це або ім'я файла-пристрою, наприклад номер або console, або просто номер, якщо ім'я файла починається з tty.

-*p* – список ідентифікаторів процесів: видавати інформацію тільки про вказані процеси;

-*u* – список ідентифікаторів користувачів: видавати інформацію тільки про процеси із заданими ідентифікаторами або вхідними іменами користувачів. Ідентифікатор користувача виводиться в числовому виді, а за наявності опції -*f* – у символьному;

-*g* – список ідентифікаторів лідерів груп: виводити інформацію тільки про процеси, для яких вказані ідентифікатори лідерів груп. *Лідер групи* – це процес, номер якого ідентичний його ідентифікатору групи. *Shell*, що запускається у разі входу в систему, є стандартним прикладом лідера групи.

Під час вказівки опції -*f ps* намагається визначити ім'я команди й аргументи, з якими був створений процес, досліджуючи призначений для користувача блок процесу. Якщо це не вдається, ім'я процесу виводиться так само, як і за відсутності опції -*f*, тільки полягає в квадратних дужках.

Нижче наводяться заголовки колонок видачі і пояснюється значення їх вмісту. Букви *l* або *f* в дужках означають, що ця колонка з'являється, відповідно, у разі довгого або повного формату видачі; відсутність букв означає, що ця колонка виводиться завжди. Відмітимо, що опції -*l* і -*f* впливають тільки на формат видачі, але не на список процесів, інформація про які буде надана.

-*F(l)* – прапори (шістнадцятирічні), логічна сума яких дає такі відомості про процес:

00 – процес термінує; елемент таблиці процесів вільний;

01 – системний процес: завжди в основній пам'яті;

02 – процес трасується батьківським процесом;

04 – батьківський сигнал трасування зупинив процес; батьківський процес чекає;

08 – процес не може бути розбуджений сигналом;

10 – процес в основній пам'яті;

20 – процес в основній пам'яті; блокований до завершення події;

40 – поступає сигнал до видаленої системи;

80 – процес у черзі на введення / виведення.

-*S(l)* – статус процесу: *0* – активний, обробляється процесором; *S* – сплячий, чекає завершення події; *R* – готовий, стоїть у черзі на виконання; *I* – що народжується, процес створюється; *Z* – стан «зомбі», процес завершений, але батьківський процес не чекає цього; *T* – трасований, процес зупинений сигналом, оскільки батьківський процес трасує його; *X* – процес що зростає, чекає отримання більшого обсягу основної пам'яті;

-*UID(f, l)* – ідентифікатор власника процесу; під час вказівки опції *-f* видається вхідне ім'я користувача;

-*PID* – ідентифікатор процесу (потрібний для того, щоб термінувати процес);

-*PPID(f, l)* – ідентифікатор батьківського процесу;

-*C(f, l)* – доля виділеного планувальником часу ЦП;

-*STIME(f)* – час запуску процесу (години : хвилини : секунди). Якщо процес запущений більш ніж 24 години потому, видається місяць і день запуску;

-*PRI(l)* – пріоритет процесу; більше число означає менший пріоритет;

-*NI(l)* – поправка до пріоритету;

-*ADDR(l)* – адреса процесу в пам'яті;

-*SZ(l)* – розмір (у блоках по 512 байтів) образу процесу в пам'яті;

-*WCHAN(l)* – адреса події, на яку чекає процес. В активного процесу ця колонка порожня;

-*TTY* – термінал (зазвичай термінал, з якого був запущений процес; нижче – СЮРПРИЗИ), що управляє. Якщо такого немає, видається символ «?»;

-*TIME* – витрачений процесом час ЦП;

-*COMMAND* – ім'я програми; якщо вказана опція *-f*, то виводиться повне ім'я команди і її аргументи.

Процес, що закінчив виконання своєї програми, і батьківський процес, що не дочекався ще завершення, як ім'я програми отримує <defunct>.

Файли. */dev/kmem* – віртуальна пам'ять ядра;

/dev/mem – пам'ять;

/etc/passwd – файл паролів, з якого береться інформація про ідентифікатори користувачів;

/etc/psdata – внутрішня структура даних;

/unix – ім'я файла, звідки була завантажена операційна система.

Сюрпризи. Під час виконання команди *ps* стан речей може змінитися; отже, *ps* дає лише наближення до дійсності. Деякі відомості, що належать до процесів, відмічених як *<defunct>*, не відповідають дійсності.

Якщо не вказана жодна з опцій *-t*, *-p*, *-u*, *-g*, команда *ps* перевіряє стандартне введення, стандартне виведення і стандартний протокол (у вказаному порядку) для визначення терміналу, що управляє і намагається вивести інформацію про процеси, які асоціюються з цим терміналом. За ситуації, коли усі три стандартні напрями введення / виведення перепризначувалися, *ps* не може знайти і видати ім'я терміналу, що управляє.

Системний виклик *fork*. Назва: *fork* – створення нового процесу. Синтаксис: *int fork()*.

Опис. Виклик *fork* призводить до створення нового процесу (породженого процесу) – точної копії процесу, що зробив виклик (батьківського процесу). А точніше, породжений процес наслідує у батьківського процесу такі характеристики:

1. Оточення.
2. Прапор «закрити у разі виконання виклику *exit*».
3. Способи обробки сигналів (тобто *SIG DFL*, *SIG IGN*, *SIGHOLD*, адреси функцій обробки сигналів).
4. Дозвіл встановлювати заново діючий ідентифікатор користувача.
5. Дозвіл встановлювати заново діючий ідентифікатор групи.
6. Стан профілізації (включено / вимкнено).
7. Значення поправки до пріоритету.
8. Усі приєднані сегменти пам'яті, що розділяються.
9. Ідентифікатор групи процесів.
10. Ідентифікатор групи терміналу.
11. Поточний робочий каталог.

12. Кореневий каталог.
13. Маска режиму створення файлів.
14. Обмеження на розмір файла.

Породжений процес відрізняється від батьківського процесу таким:

1. Породжений процес має свій унікальний ідентифікатор процесу.
2. Породжений процес має інший ідентифікатор батьківського процесу, що дорівнює ідентифікатору процесу, його породжувача.
3. Породжений процес має свої власні копії батьківських дескрипторів файлів. Кожен дескриптор файла породженого процесу розділяє з відповідним батьківським дескриптором файла загальний покажчик поточної позиції у файлі.
4. Усі `semadj` значення скидаються.
5. Породжений процес не наслідуює у батьківського процесу ознак утримання в пам'яті сегмента команд, даних або усього процесу цілком.
6. Обнуляються лічильники часу, витраченого для обслуговування цього процесу (`tms utime`, `tms stime`, `tms cutime`, `tms cstime`). Відміняється запит до будильника.

6.3.1. Моніторинг процесів

Команда `ps`

Існує також команда, що відображає будь-яку інформацію про статус процесу – команда `ps` («*process status*» – *статус процесу*). Команда `ps` приймає як аргументи нуль або більше ідентифікаторів `PID` і відображає статус відповідних процесів. Якщо запустити команду `jobs` з опцією `-p`, то її вивід являтиме собою ідентифікатор лідера групи процесів кожного завдання.

Про статус фонових процесів

Якщо ви запустите команду `ps` без будь-яких опцій, то побачите список процесів, для яких ваш термінал є керуючим терміналом.

Деякі опції, включаючи *-f* (*full* – повний), *-j* (*jobs* – завдання) і *-l* (*long* – довгий), дозволяють керувати кількістю інформації, що виводиться. Якщо не вкажете ніякі ідентифікатори *PID*, то іншою корисною опцією є опція *-forest*, що відображає команди у вигляді дерева і показує, які процеси є батьківськими для інших процесів. Зокрема, можна побачити, що команди *sleep* з попередніх лістингів є дочірніми процесами сценаріїв, запущених вами у фоновому режимі.

Команда *free*

Команда *free* відображає кількість вільної пам'яті і пам'яті, що використовується комп'ютером. Після замовчування значення виводяться в кілобайтах, але можна змінити це за допомогою різних опцій: *-b* (байт), *-k* (кб), *-m* (мб), *-g* (гб). Опція *-t* виводить сумарний рядок, а опція *-s* оновлює інформацію з вказаною періодичністю (час вказується в секундах). У лістингу 9 наведені два приклади.

Команда *uptime*

Команда *uptime* виводить рядок, де вказано поточний час, час безперервної роботи системи, кількість підключених користувачів і середня завантаженість за останні 1, 5 і 15 хвилин.

Використання команди *top*

Якщо бажаєте відслідковувати зміни і для цього запускаєте команду *ps* декілька разів, то, можливо, кращим рішенням буде команда *top*. Ця команда відображає список процесів, що постійно оновлюється, а також корисну сумарну інформацію.

Виведення інформації про процеси за допомогою *top*

Команда *top* має декілька підкоманд, з яких найбільш корисними є:

h – виведення довідкової інформації; *q* – завершення роботи команди *top*; *f* – додавання і видалення полів, що відображаються; *o* – сортування порядку виведення інформації; *F* – вибір полів, за якими виконується сортування.

6.3.2. Потоки, програмні канали перенаправлення в ОС Linux

Перенаправлення стандартного введення / виведення

Командний інтерпретатор Linux, такий як *bash*, отримує вхідні дані та направляє вихідні дані у вигляді послідовностей або потоків символів. Будь-який символ не залежить ні від попередніх, ні від наступних символів. Символи не впорядковані у вигляді структурованих записів або блоків з фіксованим розміром. Доступ до потоків здійснюється за допомогою механізмів введення / виведення незалежно від того, звідки надходять і куди передаються потоки символів (файл, клавіатура, вікно, екран або інший пристрій введення / виведення). Командні інтерпретатори Linux використовують три стандартні потоки введення / виведення, кожному з яких назначений певний файловий дескриптор:

stdout – стандартний потік виведення, відображає виведення команд і має дескриптор;

stderr – стандартний потік помилок, відображає помилки команд і має дескриптор 2;

stdin – стандартний потік введення, передає вхідні дані командам і має дескриптор 0.

Потоки введення забезпечують передачу вхідних даних (зазвичай тих, що надходять з клавіатури) командам. Потоки виведення забезпечують друк текстових символів, як правило, на термінал. Спочатку термінал являв собою ASCII друкуючий пристрій або дисплейний термінал, але нині, як правило, це просто вікно робочого столу комп'ютера.

Перенаправлення виведення

Існує два способи перенаправлення виведення в файл:

***n*>** – перенаправляє виведення із файлового дескриптора *n* у файл. Ви повинні мати права на запис у файл. Якщо файл не існує, то він буде створений. Якщо файл існує, то весь його вміст, як правило, знищується без будь-яких попереджень;

***n*>>** – також перенаправляє виведення із файлового дескриптора *n* у файл. Ви також повинні мати права на запис у файл.

Якщо файла не існує, то він буде створений. Якщо файл існує, то виведення додається до його вмісту.

Символ n в операторах $n>$ або $n>>$ є файловим дескриптором. Якщо він не вказаний, то припускається, що використовується стандартний пристрій виведення.

Перенаправлення виведення за допомогою оператора $n>$ зазвичай призводить до перезапису існуючих файлів. Ви можете керувати цією властивістю за допомогою опції *noclobber* вбудованої команди *set*. Якщо ця опція визначена, то можете перевизначити її за допомогою оператора $n>|$.

Перенаправлення виведення за допомогою опції *noclobber*

Іноді може знадобитися перенаправити у файл як стандартне виведення, так і стандартний потік помилок. Часто це використовується в автоматизованих процесах або фонових завданнях для того, щоб потім можна було подивитися результати роботи. Щоб перенаправити стандартне виведення і стандартний потік помилок в одне і те саме місце, використовуйте оператор $&>$ або $&>>$. Альтернативний варіант – перенаправити файловий дескриптор n і потім файловий дескриптор m в одне і те саме місце за допомогою конструкції $m>&n$ або $m>>&n$. У цьому випадку важливий порядок перенаправлення потоків. Наприклад, команда *command 2>&1 >output.txt* – це не те саме, що команда *command >output.txt 2>&1*.

У першому випадку потік помилок *stderr* перенаправляється в поточне місце розташування потоку *stdout*, а потім потік *stdout* перенаправляється у файл *output.txt*; однак друге перенаправлення зачіпає лише *stdout*, але не *stderr*. У другому випадку потік *stderr* перенаправляється в поточне місце розташування потоку *stdout*, тобто у файл *output.txt*.

Перенаправлення двох потоків в один файл

В інших випадках вам може знадобитися повністю проігнорувати стандартне виведення або стандартний потік помилок. Для цього варто перенаправити відповідний потік у пустий файл */dev/null*.

Ігнорування стандартного потоку помилок шляхом використання */dev/null*

Перенаправлення введення

Так само, як можемо перенаправляти потоки *stdout* і *stderr*, можемо перенаправити потік *stdin* із файла за допомогою оператора *<*. Команда *tr* використовується для заміни пробілів у файлі *text1* на символи табуляції. Виведення команди *cat* можна використати для того, щоб створити стандартний потік введення для команди *tr*. Але для перетворення пробілів у символи табуляції замість даремного виклику команди *cat* можемо використати перенаправлення введення.

Перенаправлення введення

У командних інтерпретаторах, зокрема і в *bash*, реалізована концепція *here-document*, яка є одним із способів перенаправлення введення. У ній використовується конструкція *<<* і будь-яке слово, наприклад *END*, яке є маркером або сигнальною міткою, що означає кінець введення.

Перенаправлення введення з використанням концепції *here-document*

Але чому просто не можна набрати команду *sort -k2*, ввести дані і натиснути комбінацію *Ctrl-d*, яка позначає кінець введення? Зрозуміло, що ви могли б виконати цю команду, але тоді не дізналися б про концепцію *here-document*, яка досить часто використовується в сценаріях командною оболонкою (в яких не існує іншої можливості вказати, які саме рядки повинні сприйматися як введення). Оскільки для вирівнювання тексту і забезпечення зручності читання в сценаріях використовуються символи табуляції, то існує інший прийом використання концепції *here-document*. Під час використання оператора *<<-* замість оператора *<<* початкові символи табуляції видаляються.

Розділення виведення

Іноді може виникнути необхідність продивлятися виведення на екрані й одночасно зберігати його у файл. Для цього ви могли

б перенаправити виведення команди у файл в одному вікні, а потім за допомогою *tail -fn1* відслідковувати виведення в іншому вікні, однак простіше використовувати команду *tee*.

Команда *tee* використовується в конвеєрі, а її аргументом є ім'я файла (або імена декількох файлів), в який буде передаватися стандартне виведення. Опція *-a* дозволяє не заміщати старий вміст файла новим вмістом, а додавати дані в кінець файла. Якщо ви хочете зберегти як стандартне виведення, так і потік помилок, то необхідно перенаправити потік *stderr* у потік *stdout*, перш ніж передавати дані на вхід команді *tee*.

6.4. Потоки і процеси в ОС Windows

6.4.1. Створення і керування процесом в ОС Windows

Процес – це додаток, що виконується і складається із особового віртуального адресного простору, коду даних та інших ресурсів операційної системи, таких як файли, синхронізаційні об'єкти, що відомі для процесу.

Процес має складатися з використовуваного модуля, особистого адресного простору і гілки. У кожного процесу має бути хоча б одна гілка. Фактично гілка – це черга, що виконується.

Коли ОС Windows вперше створює процес, вона створює тільки одну гілку на процес, ця гілка зазвичай починає виконання з першої інструкції в модулі, якщо в подальшому знадобиться більше гілок, він може сам створювати його. Коли Windows отримає команду для створення процесу, то вона створює особистий адресний простір для процесу, а потім завантажує виконуваний файл у просторі. Після цього ОС створює основну гілку для процесу.

Далі представлено синтаксис функції створення процесу в ОС Windows `CreateProcess`:

```
CreateProcess    proto    lpApplicationName:
DWORD, \ lpComandLine:DWORD, \
    lpProcessAttributes:DWORD, \
lpThreatAttributes:DWORD, \
```

```
bInheritHandles: DWORD, \ dwCreationFlags:
DWORD, \ lpEnviroment: DWORD, \
lpCurrentDirectory: DWORD, \ lpStartupInfo:
DWORD, \ lpProcessInformation: DWORD
```

Більшу частину параметрів можна ігнорувати.

lpApplicationName – ім'я виконуючого файлу з або без шляху. Якщо параметр дорівнює нулю, маємо представити ім'я виконуючого файлу в параметрі *lpComandLine*.

lpComandLine – аргументи командного рядка до програми, яку необхідно запустити.

lpProcessAttributes і *lpThreatAttributes* – тут необхідно вказати атрибути безпеки для процесу й основної гілки. Якщо вони дорівнюють нулям (NULL), то використовуємо атрибути безпеки за замовчуванням.

bInheritHandles – прапор, який вказує, чи буде новий процес наслідувати всі відкриті *Handle* вашого процесу.

dwCreationFlags – декілька прапорів, які визначають поведінку процесу під час його створення, наприклад процес може бути створений і одразу призупинений, щоб була змога перевірити його або змінити, перш ніж він зупиниться.

lpEnviroment – вказівник на блок пам'яті, який містить декілька змінних оточень для нового процесу.

lpCurrentDirectory – вказівник на рядок, який вказує на поточний диск і директорію для дочірнього процесу. NULL, якщо дочірній процес має унаслідувати її від батьківського процесу.

lpStartupInfo – вказує на структуру *STARTUPINFO*, яка визначає, як має з'явитися основне вікно нового процесу.

lpProcessInformation – вказує на структуру *PROCESS_INFORMATION*, яка отримує ідентифікаційну інформацію про новий процес.

Структура *PROCESS_INFORMATION* має такі параметри:

PROCESS_INFORMATION STRUCT

hProcess HANDLE ?; хендл дочірнього процесу

process

hThread HANDLE ?; хендл основної гілки дочірнього процесу

dwProcessId DWORD ?; ID дочірнього процесу

dwThreadId DWORD ?; ID основної гілки

PROCESS_INFORMATION ENDS

Хендл процесу і ID процесу – це різні речі. **ID процесу** – це унікальний ідентифікатор у системі. **Хендл процесу** – це значення, що повертає Windows для використання іншими API-функціями, пов'язаними з процесами. Хендл процесу не може використовуватись для ідентифікації процесу, оскільки він неунікальний.

Після виклику функції *CreateProcess* створюється новий процес і функція відразу повертається. Можна перевірити, чи є цей процес активним, викликавши функцію *GetExitCodeProcess*, яка має такий синтаксис:

GetExitCodeProcess proto hProcess:DWORD, lpExitCode:DWORD.

Якщо виклик цієї функції успішний, *lpExitCode* міститиме код виходу з процесу, що запитується. Якщо значення *lpExitCode* дорівнює *STILL_ACTIVE*, тоді це означає, що процес усе ще запущений. Можна примусово перервати процес, викликавши функцію *TerminateProcess*.

6.4.2. Програмна реалізація потоків в ОС Windows

Потік – це ланцюг інструкцій. Завжди є можливість також створювати додаткові потоки у вашій програмі. Вважається, що мультипотокковість – це багатозадачність всередині однієї програми.

Потік – це функція, яка виконується паралельно з основною програмою. Можливо запускати декілька екземплярів однієї і тієї самої функції або запускати декілька функцій одночасно залежно від вимог програміста. Мультипотокковість властива для Win 32. Потоки виконуються у тому самому процесі, тому вони мають доступ до всіх ресурсів процесу, але кожний потік має свій власний стек, тому локальні змінні в кожному потоці приватні. Кожний потік має свій власний набір регістрів, тому коли

Windows перемикається на інший потік, попередній запам'ятовує свій стан і може відновити його, коли основа отримує контроль. Це забезпечується внутрішніми засобами Windows.

Потоки поділяються на дві категорії:

- **потік інтерфейсу користувача:** потік такого типу створює своє власне вікно, тому він отримує двійкові сповіщення. Він може відповідати користувачу за допомогою свого вікна;
- **робочі потоки:** цей тип потоків не створює вікна, тому він не може приймати будь-які Windows-сповіщення та існує тільки для того, щоб виконувати призначену йому роботу на задньому фоні відповідно до його призначення.

Найкраща стратегія під час використання мультипотоків властивостей Win 32 – це дозволити основному потоку робити все, що пов'язане з користувацьким інтерфейсом, а іншим виконувати роботу у фоновому режимі. У цьому випадку основний потік – головний, інші потоки – його помічники.

Можемо створити потік за допомогою виклику функції *CreateThread*, яка має такий синтаксис:

```
CreateThread                                     proto
LpThreadAttributes:DWORD,\
    dwStackSize:      DWORD,\      LpStartAddress:
DWORD,\LpParameter:      DWORD,\dwCreationFlags:
DWORD,\ LpThreadId:  DWORD
```

Функція *CreateThread* подібна до *CreateProcess*.

lpThreadAttributes – можливо використовувати NULL, якщо необхідно, щоб у потоці були установки безпеки за замовчуванням.

dwStackSize – вказати розмір стека потоку. Якщо необхідно, щоб потік мав такий самий розмір стека, як і в основного, використовують NULL як параметр.

lpStartAddress – адреса функції потоку. Ця функція буде виконувати призначену для потоку роботу. Ця функція має отримувати один і тільки один 32-бітний параметр і повертати 32-бітне значення.

lpParametr – параметр, який необхідно передати функції потоку.

dwCreationFlags – 0 означає, що потік починає виконуватися відразу після створення. Для зворотного можна використовувати прапор *CREATE_SUSPEND*.

lpThreadId – *CreateThread* – помістити сюди *ID* створеного потоку.

Якщо виклик *CreateThread* пройшов успішно, вона повертає хендл створеного потоку, в іншому випадку вона повертає *NULL*.

Функція потоку запускається так швидко, як тільки закінчується виклик *CreateThread*, якщо тільки не позначено прапор *CREATE_SUSPEND*. У цьому випадку потік буде заморожений до виклику функції *ResumeThread*.

Коли функція потоку повертається (за допомогою інструкції *ret*), Windows опосередковано викликає *ExitThread* для функції потоку.

Можливо отримати код виходу потоку за допомогою функції *GetExitCodeThread*.

Існує три методи комунікації між потоками:

1. Використання глобальних змінних.
2. Windows-повідомлення.
3. Події.

Потоки розглядають ресурси процесу, включаючи глобальні змінні, тому потоки можуть використовувати їх для того, щоб взаємодіяти один з одним. Але все-таки цей метод має використовуватись обережно. Синхронізацію потрібно уважно спланувати.

Також можливо використовувати Windows-повідомлення, щоб виконувати взаємодію між потоками.

Якщо всі потоки мають користувацький інтерфейс, то немає проблем: цей метод може використовуватися для двосторонньої комунікації. Усе, що потрібно зробити, – це визначити один або більше додаткових Windows-повідомлень, які будуть використовуватися з потоками. Визначаємо повідомлення, використовуючи значення *WM_USER* як базове, наприклад:

WM_MYCUSTOMMSG *equ* *WM_USER+100h*.

Якщо один із потоків має інтерфейс користувача, а інший є робочим, не можемо використовувати цей метод для двостороннього користування, оскільки у робочого потоку немає свого вікна, а, відповідно, і черги повідомлень. Можливо використовувати такі схеми:

Потік з інтерфейсом користувача → глобальна змінна → робочий потік.

Робочий потік → Windows-повідомлення → Потік з користувацьким інтерфейсом.

Контрольні питання

1. Дайте визначення операційній системі. Які основні функції операційної системи?
2. Як класифікуються ОС за призначенням?
3. З яких компонентів складається типова ОС?
4. Які основні принципи побудови ОС ієрархічної структури?
5. Що таке програмне середовище?
6. Що таке операційне середовище?
7. Що таке API ОС?
8. Якими командами в Linux створюється каталог, вилучається каталог, здійснюється перехід від одного каталогу до іншого, копіюється каталог?
9. Якими командами в Linux створюється файл, вилучається файл, здійснюється перегляд файлів у каталозі, перейменовується файл, копіюється файл?
10. Якою командою змінюються атрибути файла в Linux?
11. Які сервіси надає типова операційна система?
12. Для ЕОМ якого покоління були розроблені перші ОС?
13. Як класифікуються ОС за режимом обробки задач?
14. Що таке Perl-скрипт?
15. Які керуючі структури є в Perl?
16. Які команди роботи з файлами і каталогами в Perl використано в Perl-скрипті зробленої практичної роботи?

17. Які регулярні вирази Perl використано в Perl-скрипті зробленої практичної роботи?
18. Що таке процеси? Які основні причини створення нових процесів в обчислювальних системах?
19. Що називається образом процесів? Назвіть атрибути процесів.
20. Які дії виконує ОС під час створення процесів?
21. Які дії виконує ОС під час перемикання процесів?
22. Які є види планування процесів?
23. Назвіть основні критерії короткострокового планування.
24. Які ви знаєте стратегії короткострокового планування процесів?
25. Дайте визначення потокам. Що таке багатопотоковість?
26. Які основні операції з потоками?
27. Назвіть і опишіть способи реалізації потоків.
28. Які елементи сумісно використовуються потоками?
29. Які елементи окремо використовуються потоками?
30. Що таке процес в ОС?
31. Що включають у себе атрибути процесу?
32. Які існують переходи між станами процесів?
33. Що містить образ процесу?
34. Які дії виконує ОС у разі створення нового процесу?
35. Які існують підходи до розробки ОС?
36. Що таке потік?
37. Назвіть елементи, що використовуються потоками одночасно?
38. Назвіть елементи, що використовуються потоками нарізно?
39. Які існують способи реалізації потоків?
40. Які переваги реалізації потоків у просторі користувача?
41. В яких ОС можлива реалізація паралельного виконання команд різних потоків?

РОЗДІЛ 7

СУЧАСНИЙ x86 ASSEMBLER.

АРХІТЕКТУРА ЯДРА x86-64

7.1. Сучасний x86 Assembler. Архітектура ядра x86-64

7.1.1. Внутрішня архітектура процесора x86-64

З точки зору прикладного програмування, внутрішня архітектура ядра процесора x86-64 може бути логічно розділена на декілька окремих виконавчих блоків, як показано на рисунку 7.1.

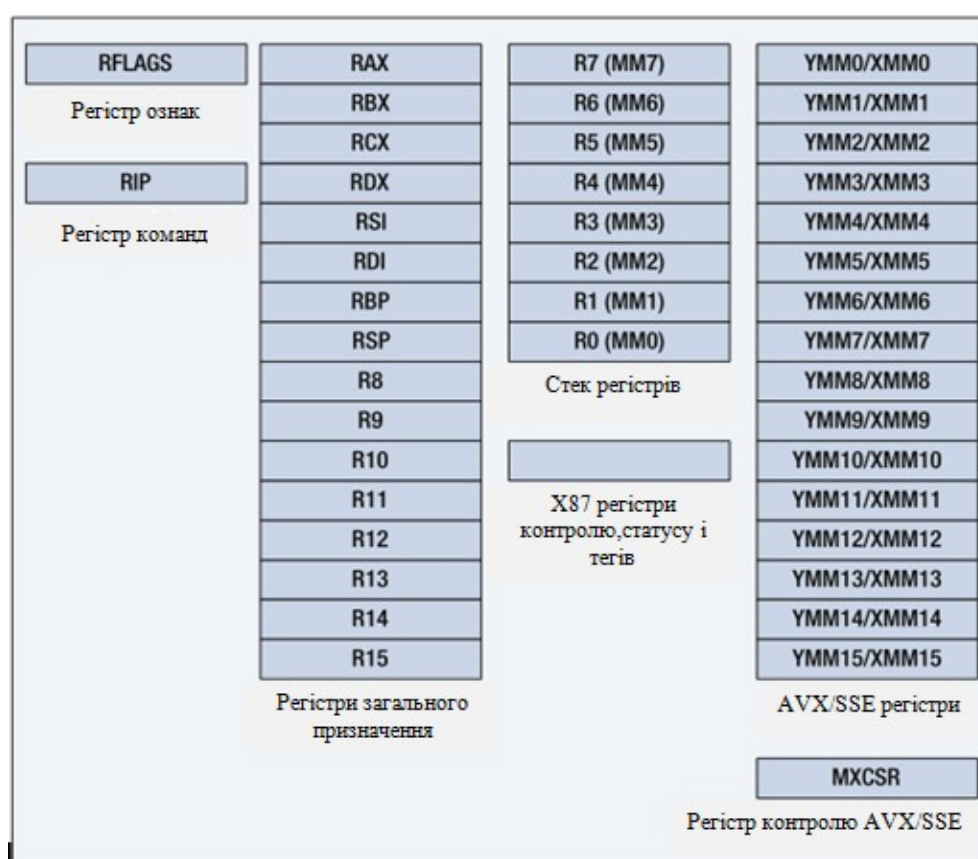


Рис. 7.1. Архітектура ядра x86-64

Основний виконавчий блок включає в себе регістри загального призначення, регістр ознак RFLAGS і регістр команд RIP (instruction pointer). Інші блоки включають у себе математичний співпроцесор x87 FPU і обчислювальні елементи,

які виконують операції SIMD (single instruction, multiple data – одиночний потік команд, множинний потік даних). Під час виконання інструкцій завжди використовуються ресурси основного виконавчого блоку, використання ресурсів FPU або SIMD не є обов’язковим.

7.1.2. Регістри загального призначення

Основний виконавчий блок x86-64 містить 16 64-розрядних регістрів загального призначення, які використовуються для виконання арифметичних, логічних і адресних операцій. Вони також можуть містити вказівники для посилення на елементи даних у пам’яті. Нижнє подвійне слово, слово і байт кожного 64-бітного регістра є незалежними один від одного і можуть бути використані для роботи з 32-бітними, 16-бітними і 8-бітними операндами. На рисунку 7.2 показано повний набір регістрів загального призначення.

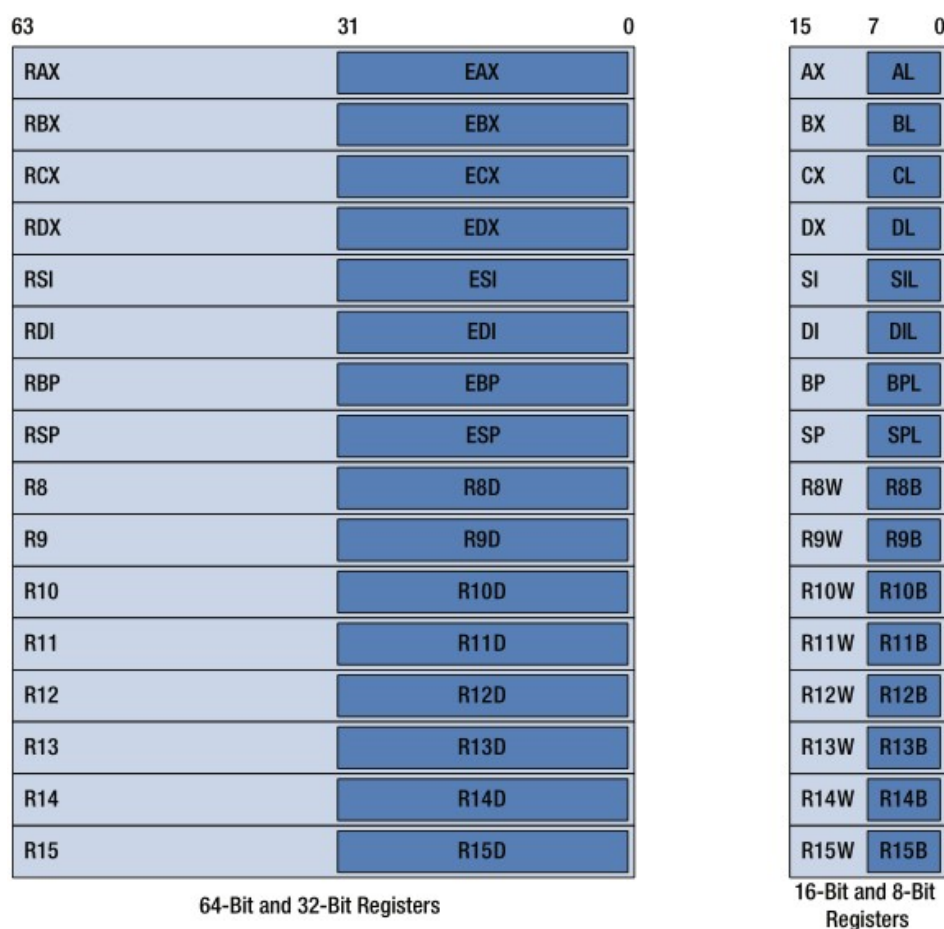


Рис. 7.2. Регістри загального призначення ядра x86-64

7.1.3. Інструкції, операнди

Більшість x86-64 інструкції використовують операнди, які позначають конкретні значення, над якими виконуються дії. Майже всі команди вимагають один операнд призначення, який міститиме результат, і один або більше вхідних операндів (операнд джерела). Більшість інструкцій вимагає чіткого завдання операндів джерела і призначення. Проте ряд x86-64 інструкцій використовують неявні і скорочені операнди. X86-64 режим підтримує ті самі три основних типи операндів, що і x86-32 режим: константа, регістр і пам'ять. У таблиці 7.1 наведені приклади команд, які ілюструють різні типи операндів.

Таблиця 7.1

Приклади базових типів операндів

Тип	Приклад	Еквівалент C/C++
Константа	mov rax,42	rax = 42
	imul r12,-47	r12 *= -47
	shl r15,8	r15 <<= 8
	xor ecx,80000000h	ecx ^= 0x80000000
	sub r9b,14	r9b -= 14
Регістр	mov rax,rbx	rax = rbx
	add rbx,r10	rbx += r10
	mul rbx	rdx:rax = rax * rbx
	and r8w,0fff00h	r8w &= 0xff00
Пам'ять	mov rax,[r13]	rax = *r13
	or rcx,[rbx+rsi*8]	rcx = *(rbx+rsi*8)
	mov qword ptr [r8],17	*(long long*)r8 = 17
	shl word ptr [r12],2	*(short*)r12 <<= 2

7.1.4. Режими адресації пам'яті

Набір x86-64 інструкцій вимагає чотири окремих компоненти, щоб вказати розташування операнди в пам'яті. Вони включають у себе: постійну величину зсуву, базовий регістр, регістр індексу і масштабний коефіцієнт. Використовуючи ці компоненти, процесор обчислює реальну адресу операнди в пам'яті так:

$$\text{EffectiveAddress} = \text{BaseReg} + \text{IndexReg} * \text{ScaleFactor} + \text{Disp.}$$

Процес обчислення адреси в процесорах x86-64 подібний до процесу обчислення x86-32 адреси. Базовим регістром (BaseReg) може бути будь-який регістр загального призначення. Індексним регістром (IndexReg) може бути будь-який регістр загального призначення, крім RSP. Допустимі масштабні коефіцієнти (ScaleFactor) включають у себе 1, 2, 4 і 8. Нарешті, зміщення (Disp) є константою 8-бітового, 16-бітового або 32-бітового знакового зміщення, яке кодується в інструкції. Таблиця 7.2 ілюструє різні x86-64 режими адресації пам'яті з використанням різних форм запису. Зверніть увагу, що не потрібно чітко вказувати всі компоненти, необхідні для обчислення адреси. Остаточний розмір обчисленої адреси завжди 64 біти.

Таблиця 7.2

Приклади базових типів операндів

Форма адресації	Приклад
RIP + Disp	mov rax,[Val]
BaseReg	mov rax,[rbx]
BaseReg + Disp	mov rax,[rbx+16]
IndexReg * SF + Disp	mov rax,[r15*8+48]
BaseReg + IndexReg	mov rax,[rbx+r15]
BaseReg + IndexReg + Disp	mov rax,[rbx+r15+32]
BaseReg + IndexReg * SF	mov rax,[rbx+r15*8]
BaseReg + IndexReg * SF + Disp	mov rax,[rbx+r15*8+64]

7.1.5. Відмінності між архітектурами x86-64 та x86-32

Більшість існуючих інструкції x86-32 мають x86-64 еквівалентну інструкцію, яка включає функцію використання 64-бітових адрес і операндів. Функція x86-64 може також виконувати обчислення, використовуючи інструкції, які опрацьовують 8-бітові, 16-бітові або 32-бітові регістри й операнди. Винятком є інструкція mov, для якої максимальний розмір константи в архітектурі x86-64, обмежений 32 бітами. Якщо команда використовує 64-бітний регістр або адресу пам'яті, будь-яке задане 32-бітове безпосереднє значення перед використанням буде розширене до 64 бітів.

Обмеження розміру констант обумовлено тим, що це впливає на послідовності команд, які програма повинна використовувати для виконання певних операцій. Наприклад, інструкція `mov rax,100` завантажує безпосереднє значення в регістр RAX. Машинний код використовує тільки 32 біти для кодування значення 100, але це значення розширюється до 64 бітів, перш ніж зберігається в RAX.

7.2. Огляд набору інструкцій (команд) процесора x86-64

7.2.1. Використання основних інструкцій процесора x86-64

Набір команд x86-64 є логічним продовженням набору команд x86-32. Більшість x86-32 інструкції було розвинено для підтримки 64-бітних операндів. Крім того, 64-розрядні функції, як і раніше, можуть використовувати інструкції з 8-бітними, 16-бітними і 32-бітними операндами, як показано в таблиці 7.3. Зверніть увагу, що звернення до пам'яті в цих прикладах виконується за допомогою 64-розрядних регістрів, що необхідно для того, щоб отримати доступ до всього 64-розрядного ефективного адресного простору. Коли можна в режимі x86-64 посилатися на операнд у пам'яті, використовуючи 32-бітний регістр (наприклад, `mov r10, [EAX]`), операнд повинен розташовуватися в нижніх 4 Гб 64-бітного ефективного адресного простору. Використання 32-бітних регістрів для доступу до операндів у пам'яті в режимі x86-64 не рекомендується, оскільки він вводить непотрібні перетворення й ускладнює тестування програмного забезпечення та налагодження.

Таблиця 7.3

Приклади x86-64 інструкцій з використанням операндів різного розміру

8 біт	16 біт	32 біти	64 біти
<code>add al,bl</code>	<code>add ax,bx</code>	<code>add eax,ebx</code>	<code>add rax,rbx</code>
<code>cmp dl,[r15]</code>	<code>cmp dx,[r15]</code>	<code>cmp edx,[r15]</code>	<code>cmp rdx,[r15]</code>
<code>mul r10b</code>	<code>mul r10w</code>	<code>mul r10d</code>	<code>mul r10</code>
<code>or [r8+rdi],al</code>	<code>or [r8+rdi*2],ax</code>	<code>or [r8+rdi*4],eax</code>	<code>or [r8+rdi*8],rax</code>
<code>shl r9b,cl</code>	<code>shl r9w,cl</code>	<code>shl r9d,cl</code>	<code>shl r9,cl</code>

7.2.2. Некоректні інструкції

Деякі рідко вживані x86-32 інструкції є некоректними для використання в режимі x86-64. У таблиці 7.4 перераховані ці інструкції. Зазначимо, що на початку покоління x86-64 процесори не підтримували `lahf` (Load Status Flags into AH Register) і `sahf` (Store AH into Flags) інструкції в режимі x86-64 (вони, як і раніше, працювали в режимі x86-32). На щастя, ці інструкції були відновлені та повинні бути доступні в більшості процесорів AMD і Intel з 2006 року. Програма може підтвердити підтримання процесором інструкцій `lahf` і `sahf` в режимі x86-64 шляхом тестування прапор функції `cruid LAHF / SAHF`.

Таблиця 7.4

x86-64 некоректні інструкції

Скорочення	Опис інструкції
<code>aaa</code>	ASCII корекція після додавання
<code>aad</code>	ASCII корекція після ділення
<code>aam</code>	ASCII корекція після множення
<code>aas</code>	ASCII корекція після віднімання
<code>bound</code>	Перевірка індексу з межею масиву
<code>daa</code>	Decimal корекція після додавання
<code>das</code>	Decimal корекція після віднімання
<code>into</code>	Переривання, якщо <code>EFLAGS.OF = 1</code>
<code>pora / porad</code>	Очистити всі регістри загального призначення
<code>pusha / pushad</code>	Запам'ятати всі регістри загального призначення

7.2.3. Нові інструкції процесора x86-64

Набір команд x86 включає в себе ряд нових інструкцій, які виконують операції з використанням 64-бітних операндів. Він також змінює поведінку декількох існуючих інструкцій. У таблиці 7.5 підсумовуються ці інструкції і використовуються скорочення GPR для регістрів загального призначення (General Purpose Registers).

Нові інструкції процесора x86-64

Скорочення	Опис інструкції
cdqe	Розширює 32-бітне значення з EAX і зберігає результат в RAX
cmpps	Порівнює значення в пам'яті, на які вказують регістри RSI і RDI; результатом виконання є встановлені прапори стану
cmpps	
cmpps	
cmpps	
cmpxchg16b	Порівнює RDX:RAX із 16-байтним операндом пам'яті і виконує обмін на основі результатів
cqo	Розширює вміст RAX до RDX: RAX
jrcxz	Виконує перехід на вказане місце пам'яті, якщо умова RCX == 0 виконується
lodsb	Завантажує значення з комірки пам'яті, на яку вказує регістр RSI, в AL, AX, EAX або RAX регістр
lodsw	
lods	
lods	
movsb	Копіює значення з ділянки пам'яті, на яку вказує регістр RSI, в ділянку пам'яті, на яку вказує регістр RDI
movsw	
movs	
movs	
movs	Копіює та розширює 32-бітне значення з операнда-джерела і записує його в операнд-призначення
pop	Забирає верхній елемент із стека. Ця команда копіює вміст комірки пам'яті, на яку вказує RSP, у вказаний GPR або комірку пам'яті; RSP потім автоматично коригується з урахуванням pop. Ця команда не може використовуватися з 32-бітовими операндами
popfq	Забирає верхнє четвертне слово із стека і зберігає молодші два слова в молодших 32 бітах RFLAGS. Старші 32 біти з RFLAGS встановлюються рівними нулю. Ця команда не змінює захищені біти і деякі керуючі біти в RFLAGS
push	Заносить значення GPR, комірки пам'яті або безпосереднє значення у стек; RSP автоматично коригується, щоб відобразити push. Ця інструкція не може бути використаною з 32-бітовими операндами
pushfq	Заносить RFLAGS у стек
rep	Повторює зазначену строкову команду, доки RCX != 0 і зазначена умова порівняння є істинною
rep/repz	
repne/repnz	
scasb	Порівнює значення комірки пам'яті, на яку вказує регістр RDI зі значенням, що містяться в регістрі AL, AX, EAX або RAX; встановлює прапори статусу на основі результатів порівняння
scasw	
scasd	
scasq	
stosb	Зберігає вміст регістра AL, AX, EAX або RAX у комірку пам'яті, на яку вказує регістр RDI
stosw	
stos	
stos	

7.3. Програмування ядра x86-64

7.3.1. Основи програмування x86-64

Як і випадку 32-бітного аналога, в Visual C++ x64 програмах є певні умови, яких необхідно дотримуватися у разі виклику функцій, реалізованих за допомогою x86-64 асемблера. Згідно з цими вимогами для кожного регістру загального призначення чітко вказується дозвіл (volatile) або заборона (non-volatile) модифікації його вмісту зовнішніми функціями. Це також належить і до регістрів XMM. Функція, виконана на x86-64 асемблері, може змінювати вміст будь-якого регістру, позначеного як volatile, і зберігати вміст non-volatile регістрів.

У системах, які підтримують x86-AVX, верхні 128 біт кожного YMM регістру класифікуються як volatile. Visual C++ x64 програми зазвичай не використовують математичний співпроцесор (FPU), тому функції асемблера, які використовують FPU, розцінюють весь стек регістрів як volatile.

Порівняно з 32-бітними системами в Visual C++ x64 накладаються більш суворі вимоги виклику функцій мовою асемблера. Ці вимоги варіюються залежно від того, чи є функція лінійною або нелінійною.

Характерні ознаки лінійних функцій:

- ✓ не викликає жодних інших функцій;
- ✓ не змінює вміст регістру RSP;
- ✓ не потребує виділення місця в локальному стеку;
- ✓ не змінює енергонезалежні регістри загального призначення або XMM регістри;
- ✓ не містить обробки винятків.

Лінійні функції x86-64 асемблера легше програмувати, але вони підходять тільки для відносно простих обчислювальних задач.

7.3.2. Арифметика цілих чисел

Перші чотири аргументи цілого типу або вказівники функції Visual C++ x64 записуються в регістри RCX, RDX, R8 і R9. Якщо функція містить більше 4-х вхідних аргументів, то додаткові аргументи передаються через стек. Відповідно до вимог щодо виклику функцій під зовнішню функцію виділяється 32 байти стека для використання. Це неініціалізований простір стека, який називається базовою площею (або простором), який насамперед використовується як тимчасова зона для зберігання значень аргументів, які передаються в регістрах. Базовий простір також може бути використаний для зберігання інших тимчасових значень. Варто зазначити, що 32 байти базового простору виділяється завжди, незалежно від кількості аргументів. Під час виконання операцій множення 32-бітних значень потрібно використовувати команду `movsxd` для запису результату з розширенням до 64-бітного значення, щоб уникнути переповнення. Результат виконання функції записується в регістр RAX як 64-бітне ціле значення.

7.3.3. Адресація пам'яті

Наступний приклад програми називається `MemoryAddressing`, який є прикладом використання режимів x86-64 адресації пам'яті. Лістинг 18-3 і 18-4 містять вихідний код для зразка програми `MemoryAddressing`.

Лістинг 7.1. `MemoryAddressing.cpp`

```
#include "stdafx.h"
extern "C" int NumFibVals_, FibValsSum_;
extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);
int _tmain(int argc, _TCHAR* argv[])
{
    FibValsSum_ = 0;
    for (int i = -1; i < NumFibVals_ + 1; i++)
    {
        int v1 = -1, v2 = -1, v3 = -1, v4 = -1;
        int rc = MemoryAddressing_(i, &v1, &v2, &v3, &v4);
```

```
printf("i: %2d rc: %2d - ", i, rc);
printf("v1: %5d v2: %5d v3: %5d v4: %5d\n", v1, v2, v3, v4);
}
printf("FibValsSum_ : %d\n", FibValsSum_);
return 0;
}
```

Лістинг 7.2. MemoryAddressing_.asm

```
; Simple lookup table (.const section data is read only)
.const
FibVals      dword 0, 1, 1, 2, 3, 5, 8, 13
             dword 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597
NumFibVals_  dword ($ - FibVals) / sizeof dword
public NumFibVals_
.data
FibValsSum_  dword ? ;value to demo RIP-relative addressing public FibValsSum_
.code
; extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);
;
; Description: This function demonstrates various addressing modes
; that can be used to access operands in memory.
;
; Returns:    0 = error (invalid table index)
;            1 = success
MemoryAddressing_ proc
; Make sure 'i' is valid
cmp ecx,0
jl InvalidIndex ;jump if i < 0
cmp ecx,[NumFibVals_]
jge InvalidIndex ;jump if i >= NumFibVals_
; Sign extend i for use in address calculations
movsxd rcx,ecx ;sign extend i
mov [rsp+8],rcx ;save copy of i
; Example #1 - base register
mov r11,offset FibVals ;r11 = FibVals
shl rcx,2 ;rcx = i * 4
add r11,rcx ;r11 = FibVals + i * 4
mov eax,[r11] ;eax = FibVals[i]
mov [rdx],eax ;Save to v1
; Example #2 - base register + index register
mov r11,offset FibVals ;r11 = FibVals
mov rcx,[rsp+8] ;rcx = i
shl rcx,2 ;rcx = i * 4
mov eax,[r11+rcx] ;eax = FibVals[i]
```

```

mov [r8],eax ;Save to v2
; Example #3 - base register + index register * scale factor
mov r11,offset FibVals ;r11 = FibVals
mov rcx,[rsp+8] ;rcx = i
mov eax,[r11+rcx*4];eax = FibVals[i]
mov [r9],eax ;Save to v3
; Example #4 - base register + index register * scale factor + disp
mov r11,offset FibVals-42 ;r11 = FibVals - 42
mov rcx,[rsp+8] ;rcx = i
mov eax,[r11+rcx*4+42] ;eax = FibVals[i]
mov r10,[rsp+40] ;r10 = ptr to v4
mov [r10],eax ;Save to v4
; Example #5 - RIP relative
add [FibValsSum_],eax ;Update sum
mov eax,1 ;set success return code
ret
InvalidIndex:
xor eax,eax ;set error return code
ret
MemoryAddressing_ endp
end

```

У вихідному коді файла MemoryAddressing.cpp (лістинг 7.1), функція `_tmain` містить простий цикл, який викликає функцію на асемблері `MemoryAddressing_`. Ця функція використовує значення індексу `i` для вибору значень з масиву 32-розрядних цілих чисел. Після кожного виклику `MemoryAddressing_` цілі змінні `v1`, `v2`, `v3` і `v4` містять результати доступу до масиву, використовуючи різні режими адресації. Ці значення потім відображаються за допомогою функції `PRINTF` для порівняння. Верхня частина файла `MemoryAddressing_.asm` (лістинг 7.2) описує масив з ім'ям `FibVals`. Цей масив містить набір 32-бітових цілочисельних констант, до яких здійснюватиметься доступ з використанням різних режимів адресації пам'яті. Далі йде розділ `.data`, який містить 32-розрядне ціле число з ім'ям `FibValsSum_`, яке використовується для демонстрації RIP-відносної адресації. Функція `MemoryAddressing_` починає своє виконання шляхом перевірки значення аргументу `i`, що міститься в регістрі `ECX`. Інструкція `movsxd rcx, ecx` розширює `i` від 32 до 64 бітів, оскільки

це значення використовується для обчислення адреси елемента в FibVals. Отримане значення зберігається в базовому просторі RCX у стеку для подальшого використання. Решта інструкцій функції MemoryAddressing_ показує, як отримати доступ до зазначеного елемента в FibVals з використанням різних режимів адресації пам'яті. Зверніть увагу, що в кожному з прикладів mov r11, offset FibVals, FibVals опрацьовується як 64-бітне значення.

7.3.4. Робота з цілими числами

Більшість x86-64 інструкцій може працювати з операндами розміром від 8 до 64 бітів. На прикладі програми IntegerOperands показано, як виконувати звичайні бітові логічні операції з використанням цілих чисел різного розміру. Вихідний код C++ і асемблера наведені в лістингах 7.3 і 7.4.

Лістинг 7.3. IntegerOperands.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"
// The following structure must match the structure that's
// declared in IntegerOperands_.asm.
typedef struct
{
    UInt8 a8;
    UInt16 a16;
    UInt32 a32;
    UInt64 a64;
    UInt8 b8;
    UInt16 b16;
    UInt32 b32;
    UInt64 b64;
} C1Val;
extern "C" void CalcLogical_(C1Val* cl_val, UInt8 c8[3], UInt16 c16[3],
    UInt32 c32[3], UInt64 c64[3]);
int _tmain(int argc, _TCHAR* argv[])
{
    C1Val x;
    UInt8 c8[3];
    UInt16 c16[3];
    UInt32 c32[3];
    UInt64 c64[3];
```

```

x.a8 = 0x81; x.b8 = 0x88;
x.a16 = 0xF0F0; x.b16 = 0xFF0;
x.a32 = 0x87654321; x.b32 = 0xF000F000;
x.a64 = 0x0000FFFF00000000; x.b64 = 0x0000FFFF00008888;
CalcLogical_(&x, c8, c16, c32, c64);
printf("\nResults for CalcLogical()\n");
printf("\n8-bit operations\n");
printf("0x%02X & 0x%02X = 0x%02X\n", x.a8, x.b8, c8[0]);
printf("0x%02X | 0x%02X = 0x%02X\n", x.a8, x.b8, c8[1]);
printf("0x%02X ^ 0x%02X = 0x%02X\n", x.a8, x.b8, c8[2]);
printf("\n16-bit operations\n");
printf("0x%04X & 0x%04X = 0x%04X\n", x.a16, x.b16, c16[0]);
printf("0x%04X | 0x%04X = 0x%04X\n", x.a16, x.b16, c16[1]);
printf("0x%04X ^ 0x%04X = 0x%04X\n", x.a16, x.b16, c16[2]);
printf("\n32-bit operations\n");
printf("0x%08X & 0x%08X = 0x%08X\n", x.a32, x.b32, c32[0]);
printf("0x%08X | 0x%08X = 0x%08X\n", x.a32, x.b32, c32[1]);
printf("0x%08X ^ 0x%08X = 0x%08X\n", x.a32, x.b32, c32[2]);
return 0;
}

```

Лістинг 7.4. IntegerOperands_.asm

; The following structure must match the structure that's
; declared in IntegerOperands.cpp. Note the version below
; includes "pad" bytes, which are needed to account for the
; member alignments performed by the C++ compiler.

CIVal struct

a8	byte	?
pad1	byte	?
a16	word	?
a32	dword	?
a64	qword	?
b8	byte	?
pad2	byte	?
b16	word	?
b32	dword	?
b64	qword	?

CIVal ends

.code

```

; extern "C" void CalcLogical_(CIVal* cl_val, Uint8 c8[3], Uint16 c16[3], Uint32
c32[3], Uint64 c64[3]);

```

```
;
```

; Description: The following function demonstrates logical operations
; using different sizes of integers.

```
CalcLogical_proc
; 8-bit logical operations
mov r10b,[rcx+CIVa.a8]      ;r10b = a8
mov r11b,[rcx+CIVa.b8]      ;r11b = b8
mov al,r10b
and al,r11b                  ;calc a8 & b8
mov [rdx],al
mov al,r10b
or al,r11b                   ;calc a8 | b8
mov [rdx+1],al
mov al,r10b
xor al,r11b                  ;calc a8 ^ b8
mov [rdx+2],al
; 16-bit logical operations
mov rdx,r8                   ;rdx = ptr to c16
mov r10w,[rcx+CIVa.a16]      ;r10w = a16
mov r11w,[rcx+CIVa.b16]      ;r11w = b16
mov ax,r10w
and ax,r11w                  ;calc a16 & b16
mov [rdx],ax
mov ax,r10w
or ax,r11w                   ;calc a16 | b16
mov [rdx+2],ax
mov ax,r10w
xor ax,r11w                  ;calc a16 ^ b16
mov [rdx+4],ax
; 32-bit logical operations
mov rdx,r9                   ;rdx = ptr to c32
mov r10d,[rcx+CIVa.a32]      ;r10d = a32
mov r11d,[rcx+CIVa.b32]      ;r11d = b32
mov eax,r10d
and eax,r11d                 ;calc a32 & b32
mov [rdx],eax
mov eax,r10d
or eax,r11d                  ;calc a32 | b32
mov [rdx+4],eax
mov eax,r10d
xor eax,r11d                 ;calc a32 ^ b32
mov [rdx+8],eax
; 64-bit logical operations
mov rdx,[rsp+40]              ;rdx = ptr to c64
mov r10,[rcx+CIVa.a64]        ;r10 = a64
mov r11,[rcx+CIVa.b64]        ;r11 = b64
mov rax,r10
and rax,r11                  ;calc a64 & b64
```

```

mov [rdx],rax
mov rax,r10
or rax,r11                                ;calc a64 | b64
mov [rdx+8],rax
mov rax,r10
xor rax,r11                                ;calc a64 ^ b64
mov [rdx+16],rax
ret

CalcLogical_endp
end

```

У верхній частині файлу IntegerOperands.cpp (лістинг 7.4) оголошено структуру CVal, яка містить змінні стандартних цілочисельних типів. Ця структура використовується для передачі вихідних операндів до функції CalcLogical_. Функція _tmain ініціалізує екземпляр CVal, викликає CalcLogical_ і відображає результати. Версія структури CVal на асемблері оголошується у файлі IntegerOperands_.asm (лістинг 7.4). У разі оголошення структури важливо мати на увазі, що більшість компіляторів C++ за замовчуванням вирівнює розмір структур. Компілятор мови асемблера не виконує вирівнювання структур за замовчуванням, чим пояснюються додаткові члени структури pad1 і pad2 в асемблерній версії CVal. У середині функції CalcLogical_ чотири незалежні блоки коду, які виконують побітові логічні операції з використанням різних розмірів цілочисельних операндів. Результат кожної логічної операції зберігається у відповідний результуючий масив. Ця функція також ілюструє правильне використання суфіксів, які повинні використовуватися для посилання на молодший байт, слово і подвійне слово регістрів R8–R15.

7.3.5. Арифметика з плаваючою крапкою

Усі x86-64 сумісні процесори включають скалярні ресурси з плаваючою крапкою SSE2, що дозволяє реалізувати арифметику з плаваючою крапкою за допомогою XMM регістрів замість x87 FPU. Наявність SSE2 також полегшує використання регістрів

ХММ як аргументів функції, що містять числа у форматі з плаваючою крапкою, і для повернення результатів виконання функції. Приклад функції додавання чисел з плаваючою крапкою наведено в лістингу 7.5.

Лістинг 7.5. FloatingPointArithmetic_.asm

```
.code
; extern "C" double CalcSum_(float a, double b, float c, double d, float e, double f);
;
; Description: The following function demonstrates how to access
; floating-point argument values in an x86-64 function.
CalcSum_ proc
; Sum the argument values
cvtss2sd xmm0,xmm0          ;promote a to DPFP
addsd xmm0,xmm1             ;xmm0 = a + b
cvtss2sd xmm2,xmm2          ;promote c to DPFP
addsd xmm0,xmm2             ;xmm0 = a + b + c
addsd xmm0,xmm3             ;xmm0 = a + b + c + d
cvtss2sd xmm4,real4 ptr [rsp+40] ;promote e to DPFP
addsd xmm0,xmm4             ;xmm0 = a + b + c + d + e
addsd xmm0,real8 ptr [rsp+48]  ;xmm0 = a + b + c + d + e + f
ret
CalcSum_ endp
End
```

Відповідно до вимог Visual C++ перші чотири аргументи з плаваючою крапкою передаються у функцію за допомогою регістрів ХММ0-ХММ3. Будь-які додаткові аргументи з плаваючою крапкою передаються через стек. Якщо функція вимагає комбінації цілих аргументів і аргументів з плаваючою крапкою – перші чотири значення (ціле або з плаваючою крапкою) передаються з використанням регістрів загального призначення або ХММ регістрів, інші аргументи передаються у стеку. При цьому регістри ХММ0-ХММ5 розглядаються як *volatile*, а ХММ6-ХММ15 – як *non-volatile*. Для повернення значення у форматі з плаваючою крапкою функція повинна використовувати регістр ХММ0.

7.3.6. Способи виклику підпрограм у x86-64. Робота зі стеком

Як уже зазначалося, Visual C ++ накладає жорсткі вимоги до прологу й епілогу у лінійних функціях. Згідно з вимогами до зовнішніх функцій передбачається використання додаткових директив асемблера, які використовуються асемблером для створення статичних даних, а також обробка виключень лише засобами середовища Visual C++. Переваги створення нелінійних функцій включають повне використання всіх регістрів загального призначення і ХММ регістрів, використання вказівників на кадр стека, локальні змінні стека і здатність викликати інші функції.

Розглянемо деякі ключові інструкції і директиви для роботи зі стеком.

```

Ccl_ proc  frame
; Function prolog
push rbp                ;save caller's rbp register
.pushreg rbp
sub rsp,16              ;allocate local stack space
.alloystack 16
mov rbp,rsp            ;set frame pointer
.setframe rbp,0
RBP_RA = 24             ;offset from rbp to ret addr
.endprolog              ;mark end of prolog

```

Оператор `proc` вказує на початок прологу, функцію та атрибут `frame`, повідомляє асемблеру, що функція `Ccl_` використовує вказівник на кадр стека. Він також вказує асемблеру генерувати статичну таблицю даних, яка потрібна основній програмі Visual C ++ для обробки винятків. Інструкція `push rbp` зберігає RBP регістр у стеці, оскільки функція `Ccl_` використовує цей регістр як вказівник кадру стека. Директива `.pushreg rbp` зберігає інформацію про зсув щодо інструкції `push rbp` у таблиці обробки винятків. Зауважимо, що директиви асемблера не є інструкціями, вони вказують на те, як виконувати певні дії в процесі компіляції програми. Інструкція `sub rsp,16` виділяє 16 байт стека під локальні змінні. Сама функція `Ccl_` використовує лише

8 байт цього простору, але, згідно з вимогами щодо виклику підпрограм, це значення вирівнюється до 16 байт. Директива асемблера `.allocstack 16` зберігає інформацію про виділення стека в таблицю обробки винятків. Інструкція `mov rbp, rsp` ініціалізує регістр RBP як вказівник на кадр стека, а директива `.setframe rbp, 0` повідомляє асемблеру про цю дію. Значення зсуву 0, яке вказано в директиві `.setframe`, – різниця в байтах між RSP і RBP. Директива `.endprolog` означає кінець прологу для функції `Cc1_`.

Приклад запису значення регістру в стек: `mov [rbp+RBP_RA+8], rcx`

Приклад читання значення зі стеку: `movsx rcx, byte ptr [rbp+RBP_RA+40]`

7.3.7. Використання необов'язкових¹ регістрів загального призначення

Якщо виникає потреба у використанні регістрів загального призначення, визначених як `non-volatile`, їх значення перед використанням необхідно зберегти у стек, що виконується в пролозі так:

```
push rbp
.pushreg rbp
```

Директива `.pushreg` додає інформацію про виконання інструкції `push` у таблицю обробки виключень Visual C++. Після збереження початкового стану регістрів можна продовжувати виконання основного тіла програми. Після завершення програми перед директивою `end` в епілозі необхідно знову відновити початковий стан регістрів. Для цього спочатку потрібно відновити значення регістру RSP до того стану, в якому він перебував після виконання команди `push` у пролозі. Якщо потрібно відновити значення декількох регістрів, відновлювати регістри необхідно у зворотному порядку. Вивільнення значення зі стека виконується так: `pop rbp`.

¹ Не зовсім коректне визначення, регістри, визначені як `non-volatile`, не повинні змінювати свого стану після завершення функції на `asm`, що вказує на те, що їх вміст впливає на роботу основної програми.

7.3.8. Робота з макросами

Макроси необхідні для спрощення процесу написання програми, щоб не обтяжувати її складними блоками коду, які часто використовуються. Для того щоб використовувати макроси, необхідно додати файл, який містить визначення макроса, за допомогою директиви `include`, наприклад:

```
include <MacrosX86-64.inc>
```

Приклад використання макросів, об'явлених у файлі `MacrosX86-64.inc`, для створення прологу кадру стека:

```
Cc4_      proc frame
_CreateFrame Cc4_,16,64,rbx,rsi,r12,r13,r14,r15
_SaveXmmRegs xmm6,xmm7,xmm8,xmm9
_EndProlog
```

У цьому прикладі використовуються макроси `_CreateFrame`, `_SaveXmmRegs`, `_EndProlog`.

Також для підключення функцій Visual C++ можна використати такий запис:

```
.code
extern pow:proc
```

У цьому прикладі відбувається підключення функції `pow()`, яка виконує піднесення у степінь, бібліотеки `<math.h>`. Також варто пам'ятати, що необхідно виділяти 32 байти базового простору для всіх підпрограм:

```
; Allocate home space on stack for use by pow()
sub rsp,32
```

Приклад виклику функції `pow`:

```
; Calculate bsa1 = 0.007184 * pow(ht, 0.725) * pow(wt, 0.425);
@@: movsd xmm0,real8 ptr [r12+rsi]      ;xmm0 = height
movsd xmm8,xmm0
movsd xmm1,real8 ptr [r8_0p725]
call pow                                ;xmm0 = pow(ht,0.725)
movsd xmm6,xmm0
```

Використання макросів зменшує обсяг коду програми, а також зменшує ризик виникнення помилок під час написання.

7.3.9. Робота з масивами і рядками у x86-64 асемблері

Масиви та рядки в пам'яті представлені у вигляді неперервного блоку даних, доступ до якого здійснюється за допомогою вказівників. Розглянемо основні операції під час роботи з масивами на прикладі множення двох матриць. Код програми на асемблері наведено в лістингу 7.6.

Лістинг 7.6. MatrixMul_.asm

```
include <MacrosX86-64.inc>
        .code
extern malloc:proc
; extern "C" double* MatrixMul_(const double* m1, int nr1, int nc1, const double*
m2, int nr2, int nc2);
;
; Description: The following function computes the product of two
; matrices.
MatrixMul_ proc frame
        _CreateFrame MatMul_,0,0,rbx,r12,r13,r14,r15
        _EndProlog
; Verify the matrix size values.
        movsxd r12,edx                                ;r12 = nr1
        test r12,r12
        jle Error                                     ;jump if nr1 <= 0
        movsxd r13,r8d                                ;r13 = nc1
        test r13,r13
        jle Error                                     ;jump if nc1 <= 0
        movsxd r14,dword ptr [rbp+MatMul_ OffsetStackArgs] ;r14 = nr2
        test r14,r14
        jle Error                                     ;jump if nr2 <= 0
        movsxd r15,dword ptr [rbp+MatMul_ OffsetStackArgs+8] ;r15 = nc2
        test r15,r15
        jle Error                                     ;jump if nc2 <= 0
        cmp r13,r14
        jne Error                                     ;jump if nc1 != nr2
; Allocate storage
        mov [rbp+MatMul_ OffsetHomeRCX],rcx            ;save m1
        mov [rbp+MatMul_ OffsetHomeR9],r9             ;save m2
        mov rcx,r12                                    ;rcx = nr1
        imul rcx,r15                                   ;rcx = nr1 * nc2
        shl rcx,3                                       ;rcx = nr1 * nc2 * size real8
        sub rsp,32                                       ;allocate home space
        call malloc
```

mov rbx,rbx	;rbx = ptr to m3
; Initialize source matrix pointers and row index i	
mov rcx,[rbp+MatMul_OffsetHomeRCX]	;rcx = ptr to m1
mov rdx,[rbp+MatMul_OffsetHomeR9]	;rdx = ptr to m2
xor r8,r8	;i = 0
; Initialize column index j	
Lp1: xor r9,r9	;j = 0
; Initialize sum and index k	
Lp2: xorpd xmm4,xmm4	;sum = 0;
xor r10,r10	;k = 0;
; Calculate sum += m1[i * nc1 + k] * m2[k * nc2 + j]	
Lp3: mov rax,r8	;rax = i
imul rax,r13	;rax = i * nc1
add rax,r10	;rax = i * nc1 + k
movsd xmm0,real8 ptr [rcx+rax*8]	;xmm0 = m1[i * nc1 + k]
mov r11,r10	;r11 = k;
imul r11,r15	;r11 = k * nc2
add r11,r9	;r11 = k * nc2 + j
movsd xmm1,real8 ptr [rdx+r11*8]	;xmm1 = m2[k * nc2 + j]
mulsd xmm0,xmm1	;xmm0 = m1[i * nc1 + k]
* m2[k * nc2 + j]	
addsd xmm4,xmm0	;update sum
inc r10	;k++
cmp r10,r13	
jl Lp3	;jump if k < nc1
; Save sum to m3[i * nc2 + j]	
mov rax,r8	;rax = i
imul rax,r15	;rax = i * nc2
add rax,r9	;rax = i * nc2 + j
movsd real8 ptr [rbx+rax*8],xmm4	;m3[i * nc2 + j] = sum
; Update loop counters and repeat until done	
inc r9	;j++
cmp r9,r15	
jl Lp2	;jump if j < nc2
inc r8	;i++
cmp r8,r12	
jl Lp1	;jump if i < nr1
mov rax,rbx	;rax = ptr to m3
Done: _DeleteFrame rbx,r12,r13,r14,r15	
ret	
Error: xor rax,rax	;return NULL
jmp Done	
MatrixMul_endp	
end	

Насамперед необхідно виділити ділянку пам'яті, де буде розміщено результуючу матрицю. Для цього використовується функція `malloc`. Наступним кроком необхідно ініціалізувати вказівники на початкові матриці. Ініціалізація індексів та тимчасових змінних виконується за допомогою операції `xor`. Далі відбувається сам процес обчислення добутку двох матриць і запис результатів у результуючу матрицю.

Робота з рядками нічим не відрізняється від роботи з масивами. Рядок також представляється у вигляді неперервного блоку даних, для роботи з яким необхідно мати вказівник на початок і розмір блоку. Останнім елементом рядка завжди є символ `'\0'`, що вказує на його кінець. Також для роботи з рядками є дві спеціальні інструкції: `scasw` (виконує пошук символу кінця рядка) і `movsw` (виконує переміщення рядка). Приклад програми на асемблері, що реалізує конкатенацію рядків, приведено в лістингу 7.7.

Лістинг 7.7. `ConcatStrings_.asm`

```
include <MacrosX86-64.inc>
.code
; extern "C" int ConcatStrings_(wchar_t* des, int des_size, const wchar_t*, const*
src, int src_n)
;
; Description: This function performs string concatenation using
; multiple input strings.
;
; Returns:      -1 Invalid des_size or src_n
;               n >= 0 Length of concatenated string
ConcatStrings_ proc frame
    _CreateFrame ConcatStrings_,0,0,rbx,rsi,rdi
    _EndProlog
; Make sure des_size and src_n are greater than zero
    movsxd rdx,edx                                ;rdx = des_size
    test rdx,rdx
    jle Error                                     ;jump if des_size <= 0
    movsxd r9,r9d                                 ;r9 = src_n
    test r9,r9
    jle Error                                     ;jump if src_n <= 0
; Perform required initializations
    mov rbx,rcx                                ;rbx = des
```

<pre> xor r10,r10 xor r11,r11 mov word ptr [rbx],r10w ; Repeat loop until concatenation is finished Lp1: mov rdi,[r8+r11*8] mov rsi,rdi ; Compute length of s[i] xor rax,rax mov rcx,-1 repne scasw not rcx dec rcx ; Compute des_index + src_len mov rax,r10 add rax,rcx len(src[i]) ; Is des_index + src_len >= des_size? cmp rax,rdx jge Done ; Copy src[i] to &des[des_index] (rsi already contains src[i]) inc rcx lea rdi,[rbx+r10*2] rep movsw ; Update des_index mov r10,rax ; Update i and repeat if not done inc r11 cmp r11,r9 jl Lp1 ; Return length of concatenated string Done: mov eax,r10d _DeleteFrame rbx,rsi,rdi ret ; Return error code Error: mov eax,-1 _DeleteFrame rbx,rsi,rdi ret ConcatStrings_ endp end </pre>	<pre> ;des_index = 0 ;i = 0 ;*des = '\0'; ;rdi = src[i] ;rsi = src[i] ;find '\0' ;rcx = len(src[i]) ;rax= des_index ;rax = des_index + len(src[i]) ;rcx = len(src[i]) + 1 ;rdi = &des[des_index] ;perform string move ;des_index += len(src[i]) ;is i >= src_n? ;jump if i < src_n ;eax = trunc(des_index) ;eax = error code </pre>
--	---

7.4. SIMD розширення архітектури x86-64

7.4.1. x86 – сумісні процесори: (SSE, SSE2, SSE3, AVX, AV2)

З точки зору прикладного програмування більшість відмінностей між середовищами виконання x86-SSE-64 і x86-SSE-32 незначна. Обидва середовища використовують одні і ті самі інструкції, операнди і типи даних. Усі x86-64 сумісні процесори включають у себе обчислювальні ресурси SSE2. Підтримка x86-64 процесорами розширень, таких як SSE2 (SSE3, SSSE3, SSE4.1 і SSE4.2), варіюється залежно від конкретної мікроархітектури. Прикладна програма повинна використовувати інструкцію CPUID, щоб перевірити наявність певної для використання конкретного розширення після SSE2. Подібно до x86-SSE більшість відмінностей між середовищами X86-AVX-64 і X86-AVX-32 є відносно незначна. Варто зазначити, що не всі x86-64 сумісні процесори підтримують x86-AVX. Прикладна програма повинна використовувати інструкцію CPUID, щоб перевірити, чи підтримує процесор AVX, AVX2 або будь-який x86-AVX подібний набір функцій розширення, таких як FMA.

7.4.2. Виконавче оточення (середовище) x86-AVX

Набір регістрів x86-AVX-64 містить 16 256-бітних регістрів, які називаються YMM0-YMM15. Ці регістри можуть бути використані для роботи з різними типами даних, включаючи упаковані цілі, упаковані з плаваючою крапкою, а також скалярні значення з плаваючою крапкою. Молодші 128 біт кожного регістру YMM являють собою відповідний XMM регістр. Більшість команд x86-AVX-64 можуть використовувати будь-який з XMM або YMM регістрів як операнди.

Більшість x86-AVX інструкцій може оперувати 128-бітовими або 256-бітовими значеннями з використанням XMM або YMM регістрів, відповідно. Здебільшого під час виконання інструкцій з передачі даних вирівнювання операнда в x86-AVX виконувати не потрібно, але рекомендується для оптимальної роботи.

За винятком інструкції, які потребують регістр загального призначення як операнд, набір інструкцій x86-AVX-64 в основному такий саме, як і в 32-бітному аналозі. Виконання x86-інструкцій AVX не залежить від режиму роботи (x64 або x32). Це включає в себе обнулення регістру YMM у верхніх 128 бітах, коли відповідний регістр XMM використовується як операнд, і правила обробки, які впливають на невикористовувані біти в значеннях з плаваючою крапкою. Функції мови асемблера x86-64 повинні також використовувати інструкції `vzeroupper` або `vzeroall`, щоб уникнути можливих затримок перехідного стану, які можуть виникнути у разі перемикання між x86-AVX і x86-SSE.

7.5. Програмування x86-64 SIMD

7.5.1. Програмування x86-SSE-64. Робота із зображенням

Розглянемо роботу із зображеннями на прикладі обробки зображень гістограми. Код програми показано в лістингу 7.8.

Лістинг 7.8. Sse64ImageHistogram_.asm

```
include <MacrosX86-64.inc>
        .code
extern NUM_PIXELS_MAX:dword
; extern bool Sse64ImageHistogram_(UInt32* histo, const UInt8*
pixel_buff,UInt32 num_pixels);
;
; Description: The following function builds an image histogram.
;
; Returns:    0 = invalid argument value
;            1 = success
;
; Requires: X86-64, SSE4.1
Sse64ImageHistogram_ proc frame
    _CreateFrame Sse64Ih_,1024,0,rbx,rsi,rdi
    _EndProlog
; Make sure num_pixels is valid
    test r8d,r8d
    jz Error                                ;jump if num_pixels is zero
    cmp r8d,[NUM_PIXELS_MAX]
```

```

        ja Error                                ;jump if num_pixels too big
        test r8d,1fh
        jnz Error                                ;jump if num_pixels % 32
!= 0
        ; Make sure histo & pixel_buff are properly aligned
        mov rsi,rcx                                ;rsi = ptr to histo
        test rsi,0fh
        jnz Error                                ;jump if histo misaligned
        mov r9,rdx
        test r9,0fh
        jnz Error                                ;jump if pixel_buff
misaligned
        ; Initialize local histogram buffers (set all entries to zero)
        xor rax,rax
        mov rdi,rsi                                ;rdi = ptr to histo
        mov rcx,128                                ;rcx = size in qwords
        rep stosq                                ;zero histo
        mov rdi,rbp                                ;rdi = ptr to histo2
        mov rcx,128                                ;rcx = size in qwords
        rep stosq                                ;zero histo2
        ; Perform processing loop initializations
        shr r8d,5                                ;r8d = number of pixel
blocks
        mov rdi,rbp                                ;rdi = ptr to histo2
        ; Build the histograms
        align 16                                ;align jump target
        @@: movdqa xmm0,[r9]                    ;load pixel block
        movdqa xmm2,[r9+16]                    ;load pixel block
        movdqa xmm1,xmm0
        movdqa xmm3,xmm2
        ; Process pixels 0 - 3
        pextrb rax,xmm0,0
        add dword ptr [rsi+rax*4],1                ;count pixel 0
        pextrb rbx,xmm1,1
        add dword ptr [rdi+rbx*4],1                ;count pixel 1
        pextrb rcx,xmm0,2
        add dword ptr [rsi+rcx*4],1                ;count pixel 2
        pextrb rdx,xmm1,3
        add dword ptr [rdi+rdx*4],1                ;count pixel 3
        ; Process pixels 4 - 7
        pextrb rax,xmm0,4
        add dword ptr [rsi+rax*4],1                ;count pixel 4
        pextrb rbx,xmm1,5
        add dword ptr [rdi+rbx*4],1                ;count pixel 5
        pextrb rcx,xmm0,6

```

```

    add dword ptr [rsi+rcx*4],1                ;count pixel 6
    pextrb rdx,xmm1,7
    add dword ptr [rdi+rdx*4],1                ;count pixel 7
; Process pixels 8 - 11
    pextrb rax,xmm0,8
    add dword ptr [rsi+rax*4],1                ;count pixel 8
    pextrb rbx,xmm1,9
    add dword ptr [rdi+rbx*4],1                ;count pixel 9
    pextrb rcx,xmm0,10
    add dword ptr [rsi+rcx*4],1                ;count pixel 10
    pextrb rdx,xmm1,11
    add dword ptr [rdi+rdx*4],1                ;count pixel 11
; Process pixels 12 - 15
    pextrb rax,xmm0,12
    add dword ptr [rsi+rax*4],1                ;count pixel 12
    pextrb rbx,xmm1,13
    add dword ptr [rdi+rbx*4],1                ;count pixel 13
    pextrb rcx,xmm0,14
    add dword ptr [rsi+rcx*4],1                ;count pixel 14
    pextrb rdx,xmm1,15
    add dword ptr [rdi+rdx*4],1                ;count pixel 15
; Process pixels 16 - 19
    pextrb rax,xmm2,0
    add dword ptr [rsi+rax*4],1                ;count pixel 16
    pextrb rbx,xmm3,1
    add dword ptr [rdi+rbx*4],1                ;count pixel 17
    pextrb rcx,xmm2,2
    add dword ptr [rsi+rcx*4],1                ;count pixel 18
    pextrb rdx,xmm3,3
    add dword ptr [rdi+rdx*4],1                ;count pixel 19
; Process pixels 20 - 23
    pextrb rax,xmm2,4
    add dword ptr [rsi+rax*4],1                ;count pixel 20
    pextrb rbx,xmm3,5
    add dword ptr [rdi+rbx*4],1                ;count pixel 21
    pextrb rcx,xmm2,6
    add dword ptr [rsi+rcx*4],1                ;count pixel 22
    pextrb rdx,xmm3,7
    add dword ptr [rdi+rdx*4],1                ;count pixel 23
; Process pixels 24 - 27
    pextrb rax,xmm2,8
    add dword ptr [rsi+rax*4],1                ;count pixel 24
    pextrb rbx,xmm3,9
    add dword ptr [rdi+rbx*4],1                ;count pixel 25
    pextrb rcx,xmm2,10

```

```

        add dword ptr [rsi+rcx*4],1           ;count pixel 26
        pextrb rdx,xmm3,11
        add dword ptr [rdi+rdx*4],1         ;count pixel 27
; Process pixels 28 - 31
        pextrb rax,xmm2,12
        add dword ptr [rsi+rax*4],1         ;count pixel 28
        pextrb rbx,xmm3,13
        add dword ptr [rdi+rbx*4],1         ;count pixel 29
        pextrb rcx,xmm2,14
        add dword ptr [rsi+rcx*4],1         ;count pixel 30
        pextrb rdx,xmm3,15
        add dword ptr [rdi+rdx*4],1         ;count pixel 31
        add r9,32                           ;r9 = next pixel block
        sub r8d,1
        jnz @B                               ;repeat loop if not done
; Merge intermediate histograms into final histogram
        mov ecx,32                         ;ecx = num iterations
        xor rax,rax                        ;rax = common offset
        @@: movdqa xmm0,xmmword ptr [rsi+rax] ;load histo counts
        movdqa xmm1,xmmword ptr [rsi+rax+16]
        paddb xmm0,xmmword ptr [rdi+rax]   ;add counts from histo2
        paddb xmm1,xmmword ptr [rdi+rax+16]
        movdqa xmmword ptr [rsi+rax],xmm0 ;save final result
        movdqa xmmword ptr [rsi+rax+16],xmm1
        add rax,32
        sub ecx,1
        jnz @B
        mov eax,1                          ;set success return code
Done: _DeleteFrame rbx,rsi,rdi
        ret
Error: xor eax,eax ;set error return code
        jmp Done
Sse64ImageHistogram_ endp
        end

```

Програма містить функцію з ім'ям Sse64ImageHistogram_, яка створює гістограму зображення за допомогою інструкцій x86-64 і SSE4.1. Ця функція будує дві проміжні гістограми та об'єднує їх у результуючу. Функція Sse64ImageHistogram_ починається зі створення кадру стека, використовуючи макрос _CreateFrame. Зверніть увагу, що кадр стека включає в себе 1 024 байти локальної пам'яті, які використовуються як місце для зберігання для одного з проміжних буферів гістограми.

Буфер `histo`, наданий функції як аргумент, використовується для зберігання другої проміжної та остаточної гістограм. Після макроса `_EndProlog` аргументи функцій `histo` і `pixel_buff` перевіряються для правильного вирівнювання, `num_pixels` перевіряється на коректність значення. Розрахункові значення двох проміжних буферів гістограми потім ініціалізуються нулем за допомогою інструкції `stosq`.

Основний цикл обробки функції `Sse64ImageHistogram_` у верхній частині має дві інструкції `MOVDQA`, що завантажують наступний блок з 32 пікселів у регістри `XMM0` / `XMM1` і `XMM2` / `XMM3`. Інструкція `pextrb rax,xmm0,0` витягує нульовий піксель з `XMM0` і копіює його в регістр `RAX` (старші біти `RAX` встановлені дорівнюючими нулю). Наступна інструкція `add dword ptr [rsi+rax*4],1` оновлює відповідний лічильник пікселів в першій проміжній гістограмі. Наступні дві інструкції:

```
pextrb rbx,xmm1,1
```

```
add dword ptr [rdi+rbx*4],1
```

обробляють піксель номер 1 так само з використанням другої проміжної гістограми. Цей метод піксельної обробки повторюється для всіх інших пікселів у поточному блоці.

Після завершення основного циклу обробки значення лічильника пікселів у двох проміжних гістограмах підсумовуються для створення остаточної гістограми зображення. Макрос `_DeleteFrame` очищає локальний кадр стека і відновляє раніше збережені значення регістрів загального призначення.

7.5.2. Програмування x86-SSE-64.

Робота перетворення зображень

Для реалізації деяких алгоритмів обробки зображень буває необхідно перетворити пікселі 8-бітового чорно-білого зображення з цілого числа без знака у значення з плаваючою крапкою одинарної точності, і навпаки. У лістингу 7.9 приведено приклад програми, яка дозволяє це зробити за допомогою набору команд `x86-SSE`.

Лістинг 7.9. Sse64ImageConvert_.asm

```
include <MacrosX86-64.inc>
extern NUM_PIXELS_MAX:dword

        .const

; All of these values must be aligned to a 16-byte boundary
UInt8ToFloat      real4 255.0, 255.0, 255.0, 255.0
FloatToUInt8Min   real4 0.0, 0.0, 0.0, 0.0
FloatToUInt8Max   real4 1.0, 1.0, 1.0, 1.0
FloatToUInt8Scale real4 255.0, 255.0, 255.0, 255.0

        .code

; extern "C" bool ImageUInt8ToFloat_(float* des, const UInt8* src, UInt32
num_pixels);
;
; Description: The following function converts the values in a UInt8
; pixel buffer to normalized [0.0, 1.0] SPFP.
;
; Requires: X86-64, SSE2
ImageUInt8ToFloat_ proc frame
    _CreateFrame U2F_,0,64
    _SaveXmmRegs xmm10,xmm11,xmm12,xmm13
    _EndProlog
; Make sure num_pixels is valid and pixel buffers are properly aligned
    test r8d,r8d
    jz Error                ;jump if num_pixels
    cmp r8d,[NUM_PIXELS_MAX]
    ja Error                ;jump if num_pixels too big
    test r8d,1fh
    jnz Error                ;jump if num_pixels % 32
!= 0
    test rcx,0fh
    jnz Error                ;jump if des not aligned
    test rdx,0fh
    jnz Error                ;jump if src not aligned
; Initialize processing loop registers
    shr r8d,5                ;number of pixel blocks
    movaps xmm4,xmmword ptr [UInt8ToFloat] ;xmm4 = packed 255.0f
    pxor xmm5,xmm5           ;xmm5 = packed 0
    align 16
; Load the next block of 32 pixels
    @@: movdqa xmm0,xmmword ptr [rdx]      ;xmm0 = pixel block
    movdqa xmm10,xmmword ptr [rdx+16]     ;xmm10 = pixel block
; Promote the pixel values in xmm0 from unsigned bytes to unsigned dwords
    movdqa xmm2,xmm0
    punpcklbw xmm0,xmm5
```

	punpckhbw xmm2,xmm5	;xmm2 & xmm0 = 8 word
pixels	movdqa xmm1,xmm0 movdqa xmm3,xmm2 punpcklwd xmm0,xmm5 punpckhwd xmm1,xmm5 punpcklwd xmm2,xmm5 punpckhwd xmm3,xmm5	;xmm3:xmm0 = 16 dword
pixels	; Promote the pixel values in xmm10 from unsigned bytes to unsigned dwords	
	movdqa xmm12,xmm10 punpcklbw xmm10,xmm5 punpckhbw xmm12,xmm5	;xmm12 & xmm10 = 8
word pixels	movdqa xmm11,xmm10 movdqa xmm13,xmm12 punpcklwd xmm10,xmm5 punpckhwd xmm11,xmm5 punpcklwd xmm12,xmm5 punpckhwd xmm13,xmm5	;xmm13:xmm10 = 16
dword pixels	; Convert pixel values from dwords to SPFP	
	cvt dq2ps xmm0,xmm0 cvt dq2ps xmm1,xmm1 cvt dq2ps xmm2,xmm2 cvt dq2ps xmm3,xmm3	;xmm3:xmm0 = 16 SPFP
pixels	cvt dq2ps xmm10,xmm10 cvt dq2ps xmm11,xmm11 cvt dq2ps xmm12,xmm12 cvt dq2ps xmm13,xmm13	;xmm13:xmm10 = 16
SPFP pixels	; Normalize all pixel values to [0.0, 1.0] and save the results	
	divps xmm0,xmm4 movaps xmmword ptr [rcx],xmm0 divps xmm1,xmm4 movaps xmmword ptr [rcx+16],xmm1 divps xmm2,xmm4 movaps xmmword ptr [rcx+32],xmm2 divps xmm3,xmm4 movaps xmmword ptr [rcx+48],xmm3 divps xmm10,xmm4 movaps xmmword ptr [rcx+64],xmm10 divps xmm11,xmm4 movaps xmmword ptr [rcx+80],xmm11	;save pixels 0 - 3 ;save pixels 4 - 7 ;save pixels 8 - 11 ;save pixels 12 - 15 ;save pixels 16 - 19 ;save pixels 20 - 23

238

	LP1: mov r9d,4	;num pixel quartets per
block		
	; Convert 16 float pixels to Uint8	
	LP2: movaps xmm0,xmmword ptr [rdx]	;xmm0 = pixel quartet
	movaps xmm1,xmm0	
	cmpltps xmm1,xmm6	;compare pixels to 0.0
	andnps xmm1,xmm0	;clip pixels < 0.0 to 0.0
	movaps xmm0,xmm1	;save result
	cmpnleps xmm1,xmm7	;compare pixels to 1.0
	movaps xmm2,xmm1	
	andps xmm1,xmm7	;clip pixels > 1.0 to 1.0
	andnps xmm2,xmm0	;xmm2 = pixels <= 1.0
	orps xmm2,xmm1	;xmm2 = final clipped
pixels		
	mulps xmm2,xmm5	;xmm2 = FP pixels [0.0,
255.0]		
	cvtps2dq xmm1,xmm2	;xmm1 = dword pixels [0,
255]		
	packusdw xmm1,xmm1	;xmm1[63:0] = word pixels
	packuswb xmm1,xmm1	;xmm1[31:0] = bytes pixels
	; Save the current byte pixel quartet	
	pextrd eax,xmm1,0	;eax = new pixel quartet
	psrldq xmm3,4	;adjust xmm3 for new
quartet		
	pinsrd xmm3,eax,3	;xmm3[127:96] = new
quartet		
	add rdx,16	;update src ptr
	sub r9d,1	
	jnz LP2	;repeat until done
	; Save the current byte pixel block (16 pixels)	
	movdqa xmmword ptr [rcx],xmm3	;save current pixel block
	add rcx,16 ;update des ptr	
	sub r8d,1	
	jnz LP1	;repeat until done
	mov eax,1	;set success return code
Done:	_RestoreXmmRegs xmm6,xmm7	
	_DeleteFrame	
	ret	
Error:	mov eax,eax ;set error return code	
	jmp Done	
ImageFloatToUint8_ endp		
End		

Програма містить 2 функції перетворення: `ImageUnit8ToFloat_` та `ImageFloatToUnit8_`.

Функція `ImageUnit8ToFloat_` виконує пряме перетворення цілочисельного значення у формат з плаваючою крапкою. За кожен ітераційний основний цикл обробки перетворює 32 пікселі. Метод оптимізації зображення починається з розширення значень пікселів зображення з 8 до 32 біт за допомогою серії інструкцій розпаковки x86-SSE (`punpcklbw`, `punpckhbw`, `punpcklwd` і `punpckhwd`). Потім ці значення перетворюються у формат з плаваючою крапкою, використовуючи інструкцію `cvtdq2ps`. Результуюче значення з плаваючою крапкою потім нормалізується в $[0,0, 1.0]$ і зберігається в буфері призначення.

Функція `ImageFloatToUnit8_` виконує зворотне перетворення. Внутрішній цикл цієї функції використовує інструкції `cmpltps` та `cmpnltps` разом з функціями булевої логіки, щоб обрізати значення пікселів з плаваючою крапкою, які перебувають за межами діапазону $[0,0, 1,0]$. Ті піксельні значення, які пройшли перевірку, потім перетворюються в беззнакове ціле однобайтне число, використовуючи інструкції `cvtps2dq`, `packusdw` і `packuswb`. Отриманий квартет байтів потім зберігається в `xmm3` $[127: 96]$ з використанням `pextrd`, `psrldq` і `pinsrd` інструкцій. Цей процес повторюють ще три рази. Після завершення внутрішнього циклу `XMM3` містить 16 перетворених пікселів. Цей блок пікселів потім зберігається в буфер призначення, використовуючи команду `movdqa`.

7.5.3. Векторні масиви

Набори команд x86-SSE і x86-AVX часто використовують для прискорення виконання алгоритмів, які виконують обчислення з використанням великих векторних масивів. У лістингу 7.10 наведено зразок програми, який ілюструє, як розрахувати перехресний добуток векторів, що зберігаються в масиві.

Лістинг 7.10. Sse64VectorArrays_.asm

```

include <MacrosX86-64.inc>
        .code
; This structure must match the VectorSoA structure that's
; defined in Sse64VectorArray.h
VectorSoA struct
        X      qword      ?           ;pointer to vector X
components
        Y      qword      ?           ;pointer to vector Y
components
        Z      qword      ?           ;pointer to vector Z
components
VectorSoA ends
; extern "C" bool Sse64VectorCrossProd_(VectorSoA* c, const VectorSoA* a,
const VectorSoA* b, UInt32 num_vectors);
;
; Description: The following function computes the cross products
; of the vectors in two VectorSoA instances.
Sse64VectorCrossProd2_ proc frame
        _CreateFrame Vcp2_,0,32,rbx,rsi,rdi,r12,r13,r14,r15
        _SaveXmmRegs xmm6,xmm7
        _EndProlog
; Make sure num_vectors is valid
        test r9d,r9d
        jz Error           ;jump if num_vectors == 0
        test r9d,3
        jnz Error          ;jump if num_vectors % 4
!= 0
        shr r9d,2
; Initialize vector component array pointers
        xor rax,rax ;misaligned pointer test value
        mov rbx,[rcx+VectorSoA.X]           ;rbx = ptr to c.X
        or rax,rbx
        mov rsi,[rcx+VectorSoA.Y]           ;rsi = ptr to c.Y
        or rax,rsi
        mov rdi,[rcx+VectorSoA.Z]           ;rdi = ptr to c.Z
        or rax,rdi
        mov r10,[rdx+VectorSoA.X]           ;r10 = ptr to a.X
        or rax,r10
        mov r11,[rdx+VectorSoA.Y]           ;r11 = ptr to a.Y
        or rax,r11
        mov r12,[rdx+VectorSoA.Z]           ;r12 = ptr to a.Z
        or rax,r12
        mov r13,[r8+VectorSoA.X]           ;r13 = ptr to b.X

```

```

    or rax,r13
    mov r14,[r8+VectorSoA.Y]           ;r14 = ptr to b.Y
    or rax,r14
    mov r15,[r8+VectorSoA.Z]           ;r15 = ptr to b.C
    or rax,r15
    and rax,0fh                         ;is a pointer misaligned?
    jnz Error                           ;jump if yes
    xor rax,rax                         ;rax = common array offset
    align 16
; Load the next block of four vectors
    @@: movaps xmm0,xmmword ptr [r10+rax] ;xmm0 = a.X components
    movaps xmm1,xmmword ptr [r11+rax]    ;xmm1 = a.Y components
    movaps xmm2,xmmword ptr [r12+rax]    ;xmm2 = a.Z components
    movaps xmm6,xmm1
    movaps xmm7,xmm2
    movaps xmm3,xmmword ptr [r13+rax]    ;xmm3 = b.X components
    movaps xmm4,xmmword ptr [r14+rax]    ;xmm4 = b.Y components
    movaps xmm5,xmmword ptr [r15+rax]    ;xmm5 = b.Z components
; Compute four vector cross products
; c.X[i] = a.Y[i] * b.Z[i] - a.Z[i] * b.Y[i]
; c.Y[i] = a.Z[i] * b.X[i] - a.X[i] * b.Z[i]
; c.Z[i] = a.X[i] * b.Y[i] - a.Y[i] * b.X[i]
    mulps xmm6,xmm5                     ;xmm6 = a.Y * b.Z
    mulps xmm7,xmm4                     ;xmm7 = a.Z * b.Y
    subps xmm6,xmm7                     ;xmm6 = c.X components
    mulps xmm2,xmm3                     ;xmm2 = a.Z * b.X
    mulps xmm5,xmm0                     ;xmm5 = a.X * b.Z
    subps xmm2,xmm5                     ;xmm2 = c.Y components
    mulps xmm0,xmm4                     ;xmm0 = a.X * b.Y
    mulps xmm1,xmm3                     ;xmm1 = a.Y * b.X
    subps xmm0,xmm1                     ;xmm0 = c.Z components
    movaps [rdi+rax],xmm0                ;save c.Z
    movaps [rsi+rax],xmm2                ;save c.Y
    movaps [rbx+rax],xmm6                ;save c.X
    add rax,16                           ;update array offset
    sub r9d,1
    jnz @B                               ;repeat until done
    mov eax,1                             ;set success return code
Done: _RestoreXmmRegs xmm6,xmm7
    _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
    ret
Error: xor eax,eax ;set error return code
    jmp Done
Sse64VectorCrossProd_ endp
End

```

Перехресним добутком двох векторів A і B є третій вектор C , що перпендикулярний A і B . Компоненти X , Y , Z можуть бути знайдені з такого рівняння:

$$c_x = a_y b_z - a_z b_y; c_y = a_z b_x - a_x b_z; c_z = a_x b_y - a_y b_x$$

Функція `Sse64VectorCrossProd_` обчислює добуток з використанням перехресних об'єктів `VectorSoA`. У цьому конкретному прикладі застосування `SOA` для організації компонентів вектора надає можливість виконання операції у випадковому порядку. Це також полегшує обчислення чотирьох векторних добутків паралельно. У верхній частині основного циклу обробки функція `Sse64vectorCrossProd_` завантажує кожен з `XMM` регістрів чотирма компонентами X , Y або Z . Розрахунок чотирьох векторних добутків здійснюється за рахунок використання інструкцій `mulps` і `subps`. Результати потім зберігаються в буфер `VectorSoA` з ім'ям `s`.

7.5.4. Програмування x86-AVX-64

Задача. Обчислення об'єму і площі поверхні еліпсоїда з використанням набору x86-AVX-64 інструкцій

Еліпсоїд є тривимірною фігурою, площа перетину якої є еліпс. Розмір еліпсоїда визначається довжиною трьох його півосей a , b і c . Внутрішній об'єм еліпсоїда легко обчислюється за допомогою цих довжин. Точний розрахунок площі поверхні еліпсоїда, однак, вимагає багатокомпонентних обчислень з використанням еліптичних інтегралів. Набагато простіше використати наближення, яке дав Кнуд Томсен. Нижче наведені рівняння, які використані в прикладі програми:

$$V = \frac{4}{3} \pi abc; S = 4\pi \frac{1}{3} (a^p b^p + a^p c^p + b^p c^p)^{1/p}, p \approx 1.6075$$

У лістингу 7.11 показаний приклад мовою асемблера для функції `Avx64CalcEllipsoid_`. Відразу ж після прологу функції аргументи значення a , b , c , і n копіюються в регістри `R12–R15`, відповідно. Наступний блок інструкцій завантажує необхідні значення констант у регістри `XMM12–XMM15`. Далі виконується

початкова ініціалізація. Основний цикл починає обробку шляхом завантаження значень півосей $a[i]$, $b[i]$ і $c[i]$ в регістри XMM6–XMM8, відповідно. Потім функція обчислює об'єм еліпсоїда, використовуючи інструкції `vmulsd` і `vdivsd`. Площа поверхні еліпсоїда розраховується за допомогою раніше заданого рівняння апроксимації. Зверніть увагу, що необхідні значення аргументів копіюються в регістри XMM0 і XMM1 перед кожним викликом інструкції `pow`. Також зверніть увагу, що функція `Avx64CalcEllipsoid_` зберігає значення, що повертається з `pow` у non-volatile регістрі до наступного виклику.

Лістинг 7.11. `Avx64CalcEllipsoid_.asm`

```
include <MacrosX86-64.inc>
extern pow:proc

        .const
r8_1p0    real8 1.0
r8_3p0    real8 3.0
r8_4p0    real8 4.0
r8_pi     real8 3.14159265358979323846
        .code

; extern "C" bool Avx64CalcEllipsoid_(const double* a, const double* b,const
double* c, int n, double p, double* sa, double* vol);
;
; Description: The following function calculates the surface area
; and volume of an ellipsoid
;
; Requires: x86-64, AVX
Avx64CalcEllipsoid_ proc frame
    _CreateFrame Ce_,0,144,rbx,rsi,rdi,r12,r13,r14,r15
    _SaveXmmRegs
xmm6,xmm7,xmm8,xmm9,xmm10,xmm12,xmm13,xmm14,xmm15
    _EndProlog
; Perform required register initializations. Note that non-volatile
; registers are used since this function calls the function pow().
    test r9d,r9d                                ;is n <= 0?
    jle Error                                    ;jump if yes
    mov r12,rcx                                  ;r12 = a ptr
    mov r13,rdx                                  ;r13 = b ptr
    mov r14,r8                                   ;r14 = c ptr
    mov r15d,r9d                                 ;r15 = n
    vmovsd xmm12,real8 ptr [rbp+Ce_OffsetStackArgs] ;xmm12 = p
```

```

vmovsd xmm0,real8 ptr [r8_1p0]
vdivsd xmm13,xmm0,xmm12                ;xmm13 = 1 / p
vmovsd xmm1,real8 ptr [r8_4p0]
vmulsd xmm14,xmm1,[r8_pi]                ;xmm14 = 4 * pi
vmovsd xmm15,[r8_3p0]                    ;xmm15 = 3
mov rsi,[rbp+Ce_OffsetStackArgs+8]       ;rsi = sa ptr
mov rdi,[rbp+Ce_OffsetStackArgs+16]      ;rdi = vol ptr
xor rbx,rbx                               ;rbx      =

common array offset
sub rsp,32                                ;allocate
home area for pow()
@@: vmovsd xmm6,real8 ptr [r12+rbx]        ;xmm6 = a
vmovsd xmm7,real8 ptr [r13+rbx]          ;xmm7 = b
vmovsd xmm8,real8 ptr [r14+rbx]          ;xmm8 = c
; Calculate the ellipsoid's volume
vmulsd xmm0,xmm14,xmm6                    ;xmm0 = 4 * pi * a
vmulsd xmm1,xmm7,xmm8                      ;xmm1 = b
* c;
vmulsd xmm0,xmm0,xmm1                      ;xmm0 = 4
* pi * a * b * c
vdivsd xmm0,xmm0,xmm15                      ;xmm0 = 4 * pi * a
* b * c / 3
vmovsd real8 ptr [rdi+rbx],xmm0           ;save    ellipsoid
volume
; Calculate the ellipsoid's surface area (see text for equation)
vmovsd xmm0,xmm0,xmm6                      ;xmm0 = a
vmovsd xmm1,xmm1,xmm12                      ;xmm1 = p
call pow
vmovsd xmm9,xmm9,xmm0                      ;xmm9 = pow(a,p)
vmovsd xmm0,xmm0,xmm7                      ;xmm0 = b
vmovsd xmm1,xmm1,xmm12                      ;xmm1 = p
call pow
vmovsd xmm10,xmm10,xmm0                     ;xmm10 = pow(b,p)
vmovsd xmm0,xmm0,xmm8                      ;xmm0 = c
vmovsd xmm1,xmm1,xmm12                      ;xmm1 = p
call pow
vmulsd xmm1,xmm9,xmm10                     ;xmm1 = pow(a,p) *
pow(b,p)
vmulsd xmm2,xmm9,xmm0                      ;xmm2 = pow(a,p)
* pow(c,p)
vmulsd xmm3,xmm10,xmm0                     ;xmm3 = pow(b,p) *
pow(c,p)
vaddsd xmm0,xmm1,xmm2
vaddsd xmm0,xmm0,xmm3

```

```
        vdivsd xmm0,xmm0,xmm15                ;xmm0 = bracket
sub expression
        vmovsd xmm1,xmm1,xmm13                ;xmm1 = 1 / p
        call pow                               ;xmm0 =
pow(subexpr,1/p)
        vmulsd xmm0,xmm0,xmm14                ;xmm0 = final surface area
        vmovsd real8 ptr [rsi+rbx],xmm0        ;save surface area
; Update the counter and offset value, repeat if not finished
        add rbx,8
        sub r15,1
        jnz @B
        mov eax,1                             ;set success return
code
        Done: _RestoreXmmRegs  xmm6,xmm7,xmm8,xmm9,xmm10,xmm12,xmm13,
xmm14,xmm15
        _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
        ret
        Error: xor eax,eax ;set error return code
        jmp done
Avx64CalcEllipsoid_ endp
        end
```

7.5.5. Робота з RGB зображенням

Розглянемо приклад програми, що обчислює мінімальне і максимальне значення червоного, зеленого і синього кольорів пікселя зображення RGB.

Лістинг 7.12. Avx64CalcRgbMinMax_.asm

```
include <MacrosX86-64.inc>
; 256-bit wide constants
ConstVals segment readonly align(32)
        InitialPminVal      db    32 dup(0ffh)
        InitialPmaxVal      db    32 dup(00h)
ConstVals ends

        .code
; Macro _YmmVpextrMinub
;
; Description: The following macro generates code that extracts the
; smallest unsigned byte value from register YmmSrc.
_YmmVpextrMinub macro GprDes,YmmSrc,YmmTmp
; Make sure YmmSrc and YmmTmp are different
        .erridni <YmmSrc>, <YmmTmp>, <Invalid registers>
```

```

; Construct text strings for the corresponding XMM registers
    YmmSrcSuffix SUBSTR <YmmSrc>,2
    XmmSrc CATSTR <X>,YmmSrcSuffix
    YmmTmpSuffix SUBSTR <YmmTmp>,2
    XmmTmp CATSTR <X>,YmmTmpSuffix
; Reduce the 32 byte values in YmmSrc to the smallest value
    vextracti128 XmmTmp,YmmSrc,1
    vpmminub XmmSrc,XmmSrc,XmmTmp                                ;XmmSrc = final
16 min values
    vpsrldq XmmTmp,XmmSrc,8
    vpmminub XmmSrc,XmmSrc,XmmTmp                                ;XmmSrc = final 8
min values
    vpsrldq XmmTmp,XmmSrc,4
    vpmminub XmmSrc,XmmSrc,XmmTmp                                ;XmmSrc = final 4
min values
    vpsrldq XmmTmp,XmmSrc,2
    vpmminub XmmSrc,XmmSrc,XmmTmp                                ;XmmSrc = final 2
min values
    vpsrldq XmmTmp,XmmSrc,1
    vpmminub XmmSrc,XmmSrc,XmmTmp                                ;XmmSrc = final 1
min value
    vpextrb GprDes,XmmSrc,0                                     ;mov    final    min
value to Gpr
    endm
; Macro _YmmVpextrMaxub
;
; Description: The following macro generates code that extracts the
; largest unsigned byte value from register YmmSrc.
_YmmVpextrMaxub macro GprDes,YmmSrc,YmmTmp
; Make sure YmmSrc and YmmTmp are different
    .erridni <YmmSrc>, <YmmTmp>, <Invalid registers>
; Construct text strings for the corresponding XMM registers
    YmmSrcSuffix SUBSTR <YmmSrc>,2
    XmmSrc CATSTR <X>,YmmSrcSuffix
    YmmTmpSuffix SUBSTR <YmmTmp>,2
    XmmTmp CATSTR <X>,YmmTmpSuffix
; Reduce the 32 byte values in YmmSrc to the largest value
    vextracti128 XmmTmp,YmmSrc,1
    vpmmaxub XmmSrc,XmmSrc,XmmTmp                                ;XmmSrc = final
16 max values
    vpsrldq XmmTmp,XmmSrc,8
    vpmmaxub XmmSrc,XmmSrc,XmmTmp                                ;XmmSrc = final 8
max values
    vpsrldq XmmTmp,XmmSrc,4

```

Операційні системи

```

                                vpmaxub XmmSrc,XmmSrc,XmmTmp                ;XmmSrc = final 4
max values
                                vpsrldq XmmTmp,XmmSrc,2
                                vpmaxub XmmSrc,XmmSrc,XmmTmp                ;XmmSrc = final 2
max values
                                vpsrldq XmmTmp,XmmSrc,1
                                vpmaxub XmmSrc,XmmSrc,XmmTmp                ;XmmSrc = final 1
max value
                                vpextrb GprDes,XmmSrc,0                    ;mov    final    max
value to Gpr
                                endm
                                ; extern "C" bool Avx64CalcRgbMinMax_(UInt8* rgb[3], UInt32 num_pixels,
                                UInt8 min_vals[3], UInt8 max_vals[3]);
                                ;
                                ; Description: The following function determines the minimum and maximum
                                ; pixel values of each color plane array.
                                ;
                                ; Requires: x86-64, AVX2
                                Avx64CalcRgbMinMax_ proc frame
                                    _CreateFrame CalcMinMax_,0,48,r12
                                    _SaveXmmRegs xmm6,xmm7,xmm8
                                    _EndProlog
                                ; Make sure num_pixels and the color plane arrays are valid
                                    test edx,edx
                                    jz Error                                ;jump            if
num_pixels == 0
                                    test edx,01fh
                                    jnz Error                                ;jump            if
num_pixels % 32 != 0
                                    xor rax,rax
                                    mov r10,[rcx]                            ;r10    =    R
color plane ptr
                                    or rax,r10
                                    mov r11,[rcx+8]                            ;r11    =    G    color
plane ptr
                                    or rax,r11
                                    mov r12,[rcx+16]                            ;r12    =    B    color
plane ptr
                                    or rax,r12
                                    test rax,1fh
                                    jnz Error                                ;jump if R, G, or B
misaligned
                                ; Initialize the processing loop registers
                                    shr edx,5                                ;edx = number of
pixel blocks

```

Розділ 7. Сучасний x86 Assembler. Архітектура ядра x86-64

	xor rcx,rcx	;rcx = common
array offset		
	vmovdqa ymm3,ymmword ptr [InitialPminVal]	;ymm3 = R
minimums		
	vmovdqa ymm4,ymm3	;ymm4 = G
minimums		
	vmovdqa ymm5,ymm3	;ymm5 = B
minimums		
	vmovdqa ymm6,ymmword ptr [InitialPmaxVal]	;ymm6 = R
maximums		
	vmovdqa ymm7,ymm6	;ymm7 = G
maximums		
	vmovdqa ymm8,ymm6	;ymm8 = B
maximums		
	; Scan RGB color plane arrays for packed minimums and maximums	
	@@: vmovdqa ymm0,ymmword ptr [r10+rcx]	;ymm0 = R pixels
	vmovdqa ymm1,ymmword ptr [r11+rcx]	;ymm1 = G pixels
	vmovdqa ymm2,ymmword ptr [r12+rcx]	;ymm2 = B pixels
	vpmminub ymm3,ymm3,ymm0	;update R
minimums		
	vpmminub ymm4,ymm4,ymm1	;update G
minimums		
	vpmminub ymm5,ymm5,ymm2	;update B
minimums		
	vpmaxub ymm6,ymm6,ymm0	;update R
maximums		
	vpmaxub ymm7,ymm7,ymm1	;update G
maximums		
	vpmaxub ymm8,ymm8,ymm2	;update B
maximums		
	add rcx,32	
	sub edx,1	
	jnz @B	
	; Calculate the final RGB minimum values	
	_YmmVpextrMinub rax,ymm3,ymm0	
	mov byte ptr [r8],al	;save min R
	_YmmVpextrMinub rax,ymm4,ymm0	
	mov byte ptr [r8+1],al	;save min G
	_YmmVpextrMinub rax,ymm5,ymm0	
	mov byte ptr [r8+2],al	;save min B
	; Calculate the final RGB maximum values	
	_YmmVpextrMaxub rax,ymm6,ymm1	
	mov byte ptr [r9],al	;save max R
	_YmmVpextrMaxub rax,ymm7,ymm1	
	mov byte ptr [r9+1],al	;save max G

```

        _YmmVpextrMaxub rax,ymm8,ymm1
        mov byte ptr [r9+2],al           ;save max B
        mov eax,1                       ;set success return
code
        vzeroupper
Done:   _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame r12
        ret
Error:  xor eax,eax ;set error return code
        jmp Done
Avx64CalcRgbMinMax_ endp
end

```

Відразу після сегмента ConstVals є два макроозначення: `_YmmVpextrMinub` і `_YmmVpextrMaxub`. Ці макроси генерують код, який витягує найменше або найбільше байтове значення з регістру YMM. Коди обох макросів є однаковими, за винятком використання інструкцій `vrminub` або `vrmaxub`.

Макрос `_YmmVpextrMinub` має три параметри: цільовий регістр загального призначення (`GprDes`), регістр джерело YMM (`YmmSrc`) і регістр тимчасовий YMM (`YmmTmp`). Зверніть увагу, що `YmmSrc` і `YmmTmp` повинні бути різні. Після ініціалізації йдуть інструкції, які витягують найменше значення байта із зазначеного регістру YMM. Інструкція `vextractq128 XmmTmp, YmmSrc, 1` копіює верхні 16 байтів регістру `YmmSrc` в `XmmTmp`. Інструкція `vrminub XmmSrc, XmmSrc, XmmTmp` завантажує останні 16 мінімальних значень у `XmmSrc`. Команда `vpsrldq XmmTmp, XmmSrc, 8` зсуває копію значення, яке міститься в `XmmSrc`, праворуч на вісім байт і зберігає результат у `XmmTmp`. Це полегшує використання іншої інструкції `vrminub`, яка зменшує кількість мінімальних значень байта від 16 до 8. Повторення набору інструкцій `vpsrldq` і `vrminub` відбувається доти, доки мінімальне значення не опинеться в молодшому байті `XmmSrc`. Потім інструкція `vpextrb GprDes, XmmSrc, 0` копіює остаточне мінімальне значення в зазначений регістр загального призначення.

Функція `Avx64CalcRgbMinMax_` використовує регістри YMM3-YMM5 і YMM6-YMM8 для збереження мінімального і максимального значень RGB, відповідно. Під час кожної ітерації

основного циклу обробки серія інструкцій `vrminub` і `vrmaxub` оновлює поточні RGB мінімуми і максимуми. Після завершення основного циклу обробки вищезгадані регістри YMM містять кінцеві 32 RGB мінімальні і максимальні значення пікселів. Макроси `_YmmVpextrMinub` і `_YmmVpextrMaxub` використовуються для отримання остаточних мінімального і максимального значень пікселів RGB. Ці значення потім зберігаються в зазначеному масиві результатів.

7.5.6. Інвертування матриці з використанням SSE-інструкцій

Мультиплікативна інверсія матриць визначається так. Нехай A і X представляють матриці $N \times N$. Матриця X є оберненою до A , якщо $AX = XA = I$ правильно, де I позначає одиничну матрицю $N \times N$. Варто зазначити, що не для всіх матриць існує обернена матриця. Матриця, яка не має оберненої, називається виродженою матрицею.

Приклад програми `Avx64CalcMat4x4Inv` використовує обчислювальний метод, заснований на теоремі Келі-Гамільтона, яка використовує прості матричні операції, які легко здійснити з використанням SIMD арифметики. Рисунок 7.3 визначає рівняння, необхідні для розрахунку інверсії матриці 4×4 . Зверніть увагу, що слід матриці – сума її діагональних елементів.

$$\begin{aligned}
 A^1 &= A, A^2 = AA, A^3 = AAA, A^4 = AAAA \\
 t_n &= \text{trace}(A^n) & \text{trace}(A) &= \sum_{i=0}^{n-1} a_{ii} \\
 c_1 &= -t_1 \\
 c_2 &= -\frac{1}{2}(c_1 t_1 + t_2) \\
 c_3 &= -\frac{1}{3}(c_2 t_1 + c_1 t_2 + t_3) \\
 c_4 &= -\frac{1}{4}(c_3 t_1 + c_2 t_2 + c_1 t_3 + t_4) \\
 A^{-1} &= -\frac{1}{c_4}(A^3 + c_1 A^2 + c_2 A^3 + c_3 I)
 \end{aligned}$$

Рис. 7.3. Розрахунок оберненої матриці 4×4

Програма на асемблері (лістинг 7.13) визначає макрос з ім'ям `_Mat4x4TraceYmm`, який обчислює слід матриці 4×4 у форматі значення з плаваючою крапкою. Цей макрос має на вході матрицю 4×4 , яка завантажується в регістри YMM0 (рядки 0 і 1) і YMM1 (рядки 2 і 3). Макрос використовує інструкції `vblendps`, `vperm2f128` та `vhaddps` для обчислення сліду матриці.

Функція `Mat4x4Mul` обчислює добуток двох 4×4 матриць у форматі з плаваючою крапкою. По-перше, транспонована матриця `m2` розраховується з використанням інструкцій `vunpcklps`, `vunpckhps` та `vperm2f128`. Кожен рядок транспонованої матриці потім копіюється в нижні і верхні 128 біт регістра YMM з використанням `vperm2f128` (переміщення значень з плаваючою крапкою). Кінцева матриця розраховується з використанням інструкцій `vdpps` і `vorps`.

Функція `Mat4x4Inv_` обчислює обернену матрицю. По-перше, слід матриці `t1-t4` обчислюється за допомогою функції `Mat4x4Mul` і макросу `_Mat4x4TraceYmm`. Коефіцієнти `c1-c4` обчислюються за допомогою x86-AVX арифметики з плаваючою крапкою. Коефіцієнт `c4` потім перевіряється, щоб гарантувати, що матриця не є виродженою. Після цього обчислюється обернена матриця. Зверніть увагу, що вся арифметика, необхідна для обчислення оберненої матриці, здійснюється за допомогою упакованого множення (`vmulps`) і додавання (`vaddps`).

Лістинг 7.13. `Avx64CalcMat4x4Inv_.asm`

```
include <MacrosX86-64.inc>
ConstVals segment readonly align(32) 'const'
    Vperm2Transpose  dword    0,4,1,5,2,6,3,7
    Vperm2Trace      dword    0,2,5,7,0,0,0,0
    Mat4x4I          real4    1.0, 0.0, 0.0, 0.0
                        real4    0.0, 1.0, 0.0, 0.0
                        real4    0.0, 0.0, 1.0, 0.0
                        real4    0.0, 0.0, 0.0, 1.0
    r4_SignBitMask   dword    80000000h,80000000h,80000000h,80000000h
    r4_AbsMask       dword    7fffffffh,7fffffffh,7fffffffh,7fffffffh
    r4_1p0           real4    1.0
    r4_N1p0          real4    -1.0
```

```

r4_N0p5          real4      -0.5
r4_N0p3333       real4      -0.3333333333
r4_N0p25         real4      -0.25
ConstVals ends

                .code

; _Mat4x4TraceYmm macro
;
; Description: The following macro generates code that calculates the
; trace of a 4x4 SPFP matrix in registers ymm1:ymm0.
_Mat4x4TraceYmm macro
    vblendps ymm0,ymm0,ymm1,84h                ;copy diagonals to ymm0
    vmovdqa ymm2,ymmword ptr [VpermsTrace]
    vpermps ymm1,ymm2,ymm0                      ;ymm1[127:0] = diagonals
    vhaddps ymm0,ymm1,ymm1
    vhaddps ymm0,ymm0,ymm0                      ;ymm0[31:0] = trace
    endm
; Mat4x4Mul
;
; Description: The following function computes the product of two
; 4x4 matrices.
;
; Input:      ymm1:ymm0 m1
;             ymm3:ymm2 m2
;
; Output:     ymm1:ymm0 m1 * m2
;
; Notes: In comments below, m2T denotes the transpose of matrix m2.
Mat4x4Mul proc private
; Calculate transpose of m2
                vmovdqa ymm6,ymmword ptr [VpermpsTranspose]    ;ymm6 = vperms
indices
                vunpcklps ymm4,ymm2,ymm3
                vunpckhps ymm5,ymm2,ymm3                      ;ymm5:ymm4 =
partial transpose
                vpermps ymm2,ymm6,ymm4
                vpermps ymm3,ymm6,ymm5                      ;ymm3:ymm2 =
m2T
; Copy rows of m2T to ymm*[255:128] and ymm*[127:0]
                vperm2f128 ymm4,ymm2,ymm2,00000000b          ;ymm4 = m2T.row0
                vperm2f128 ymm5,ymm2,ymm2,00010001b          ;ymm5 = m2T.row1
                vperm2f128 ymm6,ymm3,ymm3,00000000b          ;ymm6 = m2T.row2
                vperm2f128 ymm7,ymm3,ymm3,00010001b          ;ymm7 = m2T.row3
; Perform mat4x4 multiplication, rows 0 and 1
; Note that all unused vdpps destination register elements are set to zero
; ymm8[31:0] = dp(m1.row0, m2T.row0)

```

```

; ymm8[159:128] = dp(m1.row1, m2T.row0)
; ymm9[63:32] = dp(m1.row0, m2T.row1)
; ymm9[191:160] = dp(m1.row1, m2T.row1)
; ymm10[95:64] = dp(m1.row0, m2T.row2)
; ymm10[223:192] = dp(m1.row1, m2T.row2)
; ymm11[127:96] = dp(m1.row0, m2T.row3)
; ymm11[255:224] = dp(m1.row1, m2T.row3)
    vdpss ymm8,ymm0,ymm4,11110001b
    vdpss ymm9,ymm0,ymm5,11110010b
    vdpss ymm10,ymm0,ymm6,11110100b
    vdpss ymm11,ymm0,ymm7,11111000b
    vorps ymm8,ymm8,ymm9
    vorps ymm10,ymm10,ymm11
    vorps ymm0,ymm8,ymm10                                ;ymm0    =

```

rows 0 and 1

```

; Perform mat4x4 multiplication, rows 2 and 3
; ymm8[31:0] = dp(m1.row2, m2T.row0)
; ymm8[159:128] = dp(m1.row3, m2T.row0)
; ymm9[63:32] = dp(m1.row2, m2T.row1)
; ymm9[191:160] = dp(m1.row3, m2T.row1)
; ymm10[95:64] = dp(m1.row2, m2T.row2)
; ymm10[223:192] = dp(m1.row3, m2T.row2)
; ymm11[127:96] = dp(m1.row2, m2T.row3)
; ymm11[255:224] = dp(m1.row3, m2T.row3)
    vdpss ymm8,ymm1,ymm4,11110001b
    vdpss ymm9,ymm1,ymm5,11110010b
    vdpss ymm10,ymm1,ymm6,11110100b
    vdpss ymm11,ymm1,ymm7,11111000b
    vorps ymm8,ymm8,ymm9
    vorps ymm10,ymm10,ymm11
    vorps ymm1,ymm8,ymm10                                ;ymm1    =

```

rows 2 and 3

```

    ret
Mat4x4Mul endp
; extern "C" bool Mat4x4Inv_(Mat4x4 m_inv, Mat4x4 m, float epsilon, bool*
is_singular);
;
; Description: The following function computes the inverse of a 4x4
; matrix.
;
; Requires: x86-64, AVX2
;
; Notes: In the comments below, m2 = m * m, m3 = m * m * m, etc.
; Offsets of temporary values on the stack
    OffsetM2Lo    equ 0                                ;m2 rows 0 and 1

```

```

OffsetM2Hi equ 32                ;m2 rows 2 and 3
OffsetM3Lo equ 64                ;m3 rows 0 and 1
OffsetM3Hi equ 96                ;m3 rows 2 and 3
Mat4x4Inv_ proc frame
    _CreateFrame Minv_,16,160
    _SaveXmmRegs
xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
    _EndProlog
    ; Perform required initializations and validations
    test rcx,01fh
    jnz Error                    ;jump if m_inv is
misaligned
    test rdx,01fh
    jnz Error                    ;jump if m is
misaligned
    vmovaps ymm14,[rdx]
    vmovaps ymm15,[rdx+32]       ;ymm15:ymm14 =
m
    vmovss real4 ptr [rbp],xmm2   ;save epsilon for
later use
    ; Allocate 128 bytes of 32-byte aligned stack space for temp matrices
    and rsp,0fffffe0h           ;align rsp to 32-
byte boundary
    sub rsp,128                 ;alloc space for
temp matrices
    ; Compute m2
    vmovaps ymm0,ymm14
    vmovaps ymm1,ymm15           ;ymm1:ymm0 = m
    vmovaps ymm2,ymm14
    vmovaps ymm3,ymm15           ;ymm3:ymm2 = m
    call Mat4x4Mul               ;ymm1:ymm0 = m2
    vmovaps [rsp+OffsetM2Lo],ymm0
    vmovaps [rsp+OffsetM2Hi],ymm1 ;save m2
    ; Compute m3
    vmovaps ymm2,ymm14
    vmovaps ymm3,ymm15           ;ymm3:ymm2 = m
    call Mat4x4Mul               ;ymm1:ymm0 = m3
    vmovaps [rsp+OffsetM3Lo],ymm0
    vmovaps [rsp+OffsetM3Hi],ymm1 ;save m3
    vmovaps ymm12,ymm0
    vmovaps ymm13,ymm1           ;ymm13:ymm12 =
m3
    ; Compute m4
    vmovaps ymm2,ymm14
    vmovaps ymm3,ymm15           ;ymm3:ymm2 = m

```

```

        call Mat4x4Mul                                ;ymm1:ymm0 = m4
; Compute and save matrix trace values
_Mat4x4TraceYmm
        vmovss xmm10,xmm0,xmm0                        ;xmm10 = t4
        vmovaps ymm0,ymm12
        vmovaps ymm1,ymm13
        _Mat4x4TraceYmm
        vmovss xmm9,xmm0,xmm0                        ;xmm9 = t3
        vmovaps ymm0,[rsp+OffsetM2Lo]
        vmovaps ymm1,[rsp+OffsetM2Hi]
        _Mat4x4TraceYmm
        vmovss xmm8,xmm0,xmm0                        ;xmm8 = t2
        vmovaps ymm0,ymm14
        vmovaps ymm1,ymm15
        _Mat4x4TraceYmm
        vmovss xmm7,xmm0,xmm0                        ;xmm7 = t1
; Calculate the required coefficients
; c1 = -t1;
; c2 = -1.0f / 2.0f * (c1 * t1 + t2);
; c3 = -1.0f / 3.0f * (c2 * t1 + c1 * t2 + t3);
; c4 = -1.0f / 4.0f * (c3 * t1 + c2 * t2 + c1 * t3 + t4);
;
; Registers used: t1-t4 = xmm7-xmm10, c1-c4 = xmm12-xmm15
        vxorps xmm12,xmm7,real4 ptr [r4_SignBitMask]    ;xmm12 = c1
        vmulss xmm13,xmm12,xmm7                        ;c1 * t1
        vaddss xmm13,xmm13,xmm8                        ;c1 * t1 + t2
        vmulss xmm13,xmm13,[r4_N0p5]                  ;xmm13 = c2
        vmulss xmm14,xmm13,xmm7                        ;c2 * t1
        vmulss xmm0,xmm12,xmm8                        ;c1 * t2
        vaddss xmm14,xmm14,xmm0                        ;c2 * t1 + c1 * t2
        vaddss xmm14,xmm14,xmm9                        ;c2 * t1 + c1 * t2 + t3
        vmulss xmm14,xmm14,[r4_N0p3333]               ;xmm14 = c3
        vmulss xmm15,xmm14,xmm7                        ;c3 * t1
        vmulss xmm0,xmm13,xmm8                        ;c2 * t2
        vmulss xmm1,xmm12,xmm9                        ;c1 * t3
        vaddss xmm2,xmm0,xmm1                          ;c2 * t2 + c1
* t3
        vaddss xmm15,xmm15,xmm2                        ;c3 * t1 + c2 * t2 +
c1 * t3
        vaddss xmm15,xmm15,xmm10                      ;c3 * t1 + c2 * t2 +
c1 * t3 + t4
        vmulss xmm15,xmm15,[r4_N0p25]                ;xmm15 = c4
; Make sure matrix is not singular
        vandps xmm1,xmm15,[r4_AbsMask]                ;compute fabs(c4)

```

	vcomiss xmm1,real4 ptr [rbp]	;compare against
epsilon	setp al	;set al = if unordered
	setb ah	;set ah = if fabs(c4)
< epsilon	or al,ah	;al = is_singular
	mov [r9],al	;save is_singular
state	jnz Error	;jump if singular
	; Calculate m_inv = -1.0 / c4 * (m3 + c1 * m2 + c2 * m1 + c3 * I)	
	vmovaps ymm0,[rsp+OffsetM3Lo]	
	vmovaps ymm1,[rsp+OffsetM3Hi]	
	;ymm1:ymm0 = m3	
	vbroadcastss ymm12,xmm12	
	vmulps ymm2,ymm12,[rsp+OffsetM2Lo]	
	vmulps ymm3,ymm12,[rsp+OffsetM2HI]	;ymm3:ymm2 = c1
* m2		
	vbroadcastss ymm13,xmm13	
	vmulps ymm4,ymm13,[rdx]	
	vmulps ymm5,ymm13,[rdx+32]	;ymm5:ymm4 = c2
* m		
	vbroadcastss ymm14,xmm14	
	vmulps ymm6,ymm14,[Mat4x4I]	
	vmulps ymm7,ymm14,[Mat4x4I+32]	;ymm7:ymm6 = c3
* I		
	vaddps ymm0,ymm0,ymm2	
	vaddps ymm1,ymm1,ymm3	
	;ymm1:ymm0 = m3 + c1*m2	
	vaddps ymm8,ymm4,ymm6	
	vaddps ymm9,ymm5,ymm7	
	;ymm9:ymm8 = c2*m + c3*I	
	vaddps ymm0,ymm0,ymm8	
	vaddps ymm1,ymm1,ymm9	
	;ymm1:ymm0 = matrix sum	
	vmovss xmm2,[r4_N1p0]	
	vdivss xmm2,xmm2,xmm15	;xmm2 = -
1.0 / c4		
	vbroadcastss ymm2,xmm2	
	vmulps ymm0,ymm0,ymm2	
	vmulps ymm1,ymm1,ymm2	
	;ymm1:ymm0 = m_inv	
	vmovaps [rcx],ymm0	
	vmovaps [rcx+32],ymm1	;save m_inv
	mov eax,1	;set success return
code		

```
Done: vzeroupper
      _RestoreXmmRegs
xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,
      xmm14,xmm15
      _DeleteFrame
      ret
Error: xor eax,eax
      jmp Done
Mat4x4Inv_ endp
; The following functions are for software test & debug.
Mat4x4Trace_ proc
      _Mat4x4TraceYmm
      ret
Mat4x4Trace_ endp
Mat4x4Mul_ proc
      call Mat4x4Mul
      ret
Mat4x4Mul_ endp
End
```

7.6. Робота з багатоядерними архітектурами

7.6.1. Мікроархітектура процесора

Продуктивність процесора x86 в основному визначається його мікроархітектурою. Мікроархітектура процесора характеризується організацією і функціонуванням його внутрішніх апаратних компонентів, які включають у себе шини команд і даних, декодери, планувальники, виконавчі пристрої і кеш-пам'ять. Розуміння основ мікроархітектури процесора дозволяє розробнику писати більш ефективний код. Такі компанії, як AMD і Intel, регулярно створюють нові процесори на основі вдосконалених або нових мікроархітектур.

Розглянемо мікроархітектуру Haswell фірми Intel, яка використовується у процесорах Core i7, i5 та i3 серії четвертого покоління. Структурна організація і функціонування більш ранніх мікроархітектур Intel, таких як Nehalem, Sandy Bridge і Ivy Bridge (або перше, друге і третє покоління процесорів Core i7, i5 та i3), схожі на Haswell, хоча остання включає в себе значно покращену продуктивність і зниження енергоспоживання.

7.6.2. Огляд багатоядерних процесорів

Архітектурні особливості процесора на основі Haswell або будь-якої іншої сучасної мікроархітектури найкраще вивчені за допомогою структури багатоядерного процесора. На рисунку 7.4 показана спрощена блок-схема типового чотириядерного процесора на базі Haswell. Зверніть увагу, що кожне ядро процесора включає в себе кеш першого рівня (L1) команд і даних, які позначені I-Cache та D-Cache. Як випливає з їх назви, ці кеші містять інструкції і дані, до яких ядро процесора може швидко отримати доступ.

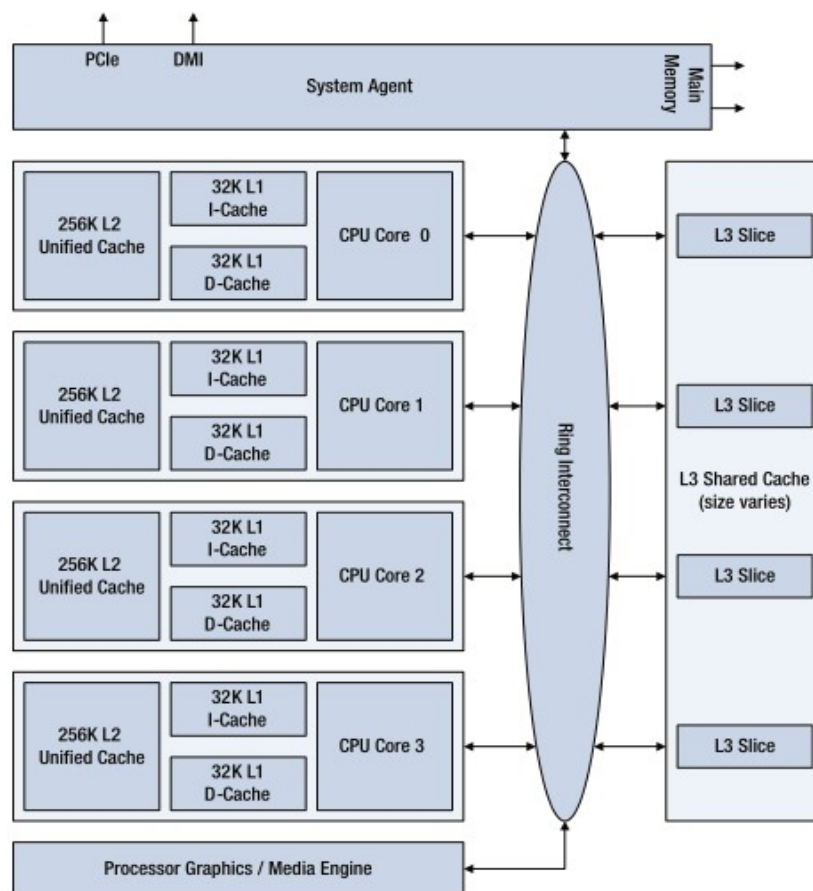


Рис. 7.4. Спрощена блок-схема чотириядерного процесора на базі Haswell

Кожне процесорне ядро також включає в себе уніфікований кеш другого рівня (L2), який містить як команди, так і дані. Кеші L1 і L2 дозволяють ядрам процесора виконувати самостійні операції паралельно, не звертаючись до загального кешу рівня L3 або оперативної пам'яті.

Якщо ядро процесора потребує інструкції або дані, яких немає в його кеш-пам'яті L1 або L2 рівня, то вони завантажуються з кешу рівня L3 або оперативної пам'яті. Кеш L3 розділяється на декілька «зрізів», кожен з яких складається з логічного контролера і масиву даних. Логічний контролер управляє доступом до відповідного масиву даних. Він також обробляє ситуації, коли потрібні дані відсутні в кеш-пам'яті і завантажує їх з основної пам'яті. Масив даних включає в себе фактичні дані кешу, які організовані в пакети по 64 байти, так звані рядки кешу. Interconnect Ring є високошвидкісною внутрішньою шиною, яка полегшує передачу даних між ядрами процесора, кешем L3, графічним модулем і системним агентом. Системний агент обробляє трафік даних між процесором, зовнішніми шинами даних та оперативною пам'яттю.

7.6.3. Мікроархітектура функціонування каналів передачі інформації

Під час виконання програми ядро процесора виконує п'ять елементарних операцій над інструкціями: захопити, декодувати, відправити, виконати і звільнити. Конкретні особливості цих операцій визначаються функціональністю каналів передачі процесорної мікроархітектури. Рисунок 7.5 показує спрощену блок-схему роботи каналів передачі центрального процесора на основі Haswell.

Блок вилучення і попереднього декодування інструкцій дістає інструкції з L1-кешу, і починається процес підготовки їх до виконання. Кроки, які виконуються на цій стадії, включають оцінку довжини інструкції, декодування x86 інструктивних префіксів, а також маркування властивостей, щоб спростити подальше декодування. Також цей блок відповідає за постійне наповнення черги команд, з якої інструкції подаються до декодерів команд. Декодери команд переводять x86 інструкції в мікрокоманди. Мікрокоманди – це самодостатні інструкції низького рівня, які виконуються виконавчим модулем.

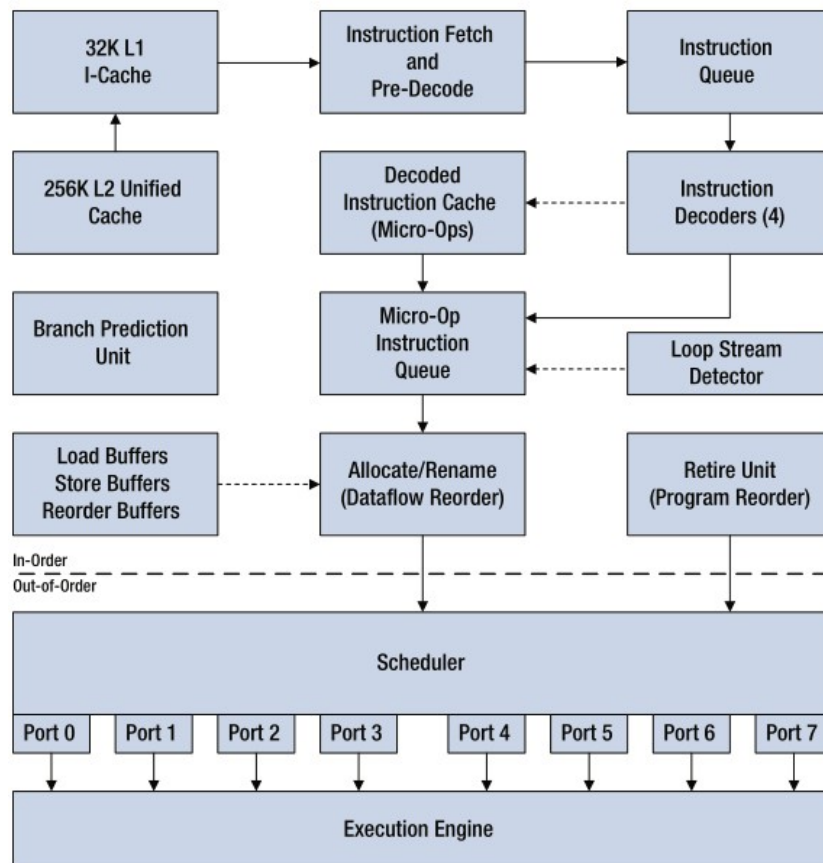


Рис. 7.5. Блок-схема роботи каналів передачі центрального процесора на основі Haswell

Мікрооперації з декодерів команд передаються до черги мікрокоманд для подальшої відправки планувальником. Вони також кешуються, за необхідності, в кеші декодованих інструкцій. Черга мікрокоманд також використовується блоком потокового пошуку петель, який ідентифікує і блокує маленькі петлі програми в черзі мікрокоманд. Це підвищує продуктивність, оскільки невеликий цикл може багаторазово виконуватись, не вимагаючи будь-яких додаткових дій.

Блок виділення / перейменування слугує мостом між каналом передачі та планувальником і виконавчим модулем. Він виділяє всі необхідні внутрішні буфери для мікрокоманд. Мікрокоманди потім передаються в планувальник. Цей пристрій формує чергу мікрооперацій, доки всі необхідні вихідні операнди не будуть доступні. Потім він відправляє готові до виконання мікрокоманди до одного з блоків виконавчого модуля. Блок вилучення видаляє мікрооперації, які завершили своє виконання. Він

також сигналізує про будь-які виняткові ситуації процесора, які можуть виникнути під час виконання мікрооперацій. І нарешті, блок передбачення допомагає вибрати наступний набір інструкцій для виконання.

7.6.4. Виконавчий модуль

Виконавчий модуль обробляє мікрооперації, передані йому планувальником. Рисунок 7.6 показує блок-схему виконавчого модуля ядра процесора на основі Haswell. Прямокутні блоки під кожним диспетчерським портом позначають виконавчий блок мікрокоманд. Зверніть увагу, що чотири з портів планувальника надають доступ до виконавчих пристроїв, які виконують функції обчислення цілих чисел, чисел з плаваючою крапкою і SIMD-арифметику. Решта чотири порти підтримують операції читання з пам'яті та зберігання.

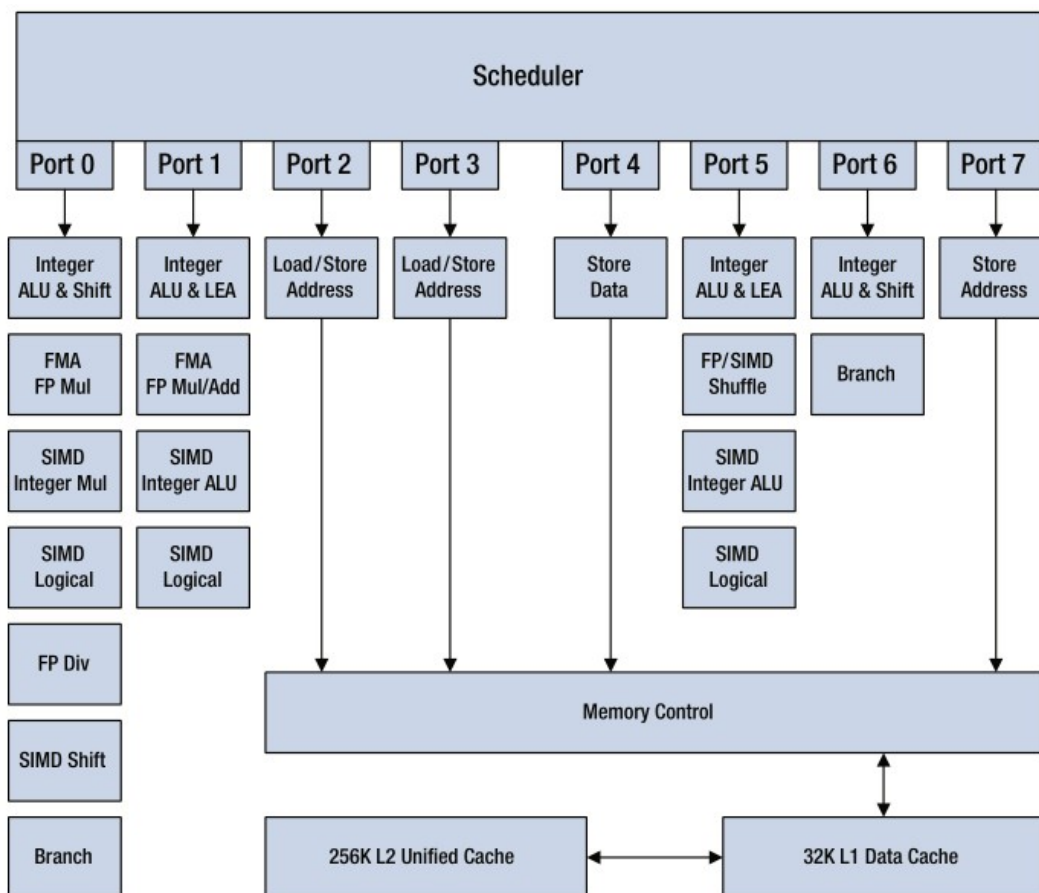


Рис. 7.6. Блок-схема виконавчого модуля ядра процесора на основі Haswell

Кожен виконавчий блок виконує певний розрахунок або операцію. Наприклад, блок цілочисельного АЛП і зсуву виконує цілочисельні арифметичні операції та операції зсуву. Блок цілочисельного АЛП SIMD призначений для реалізації цілочисельної SIMD-арифметики. Планувальник може відправити максимум вісім мікрооперацій за такт (по одному на кожен порт) до виконавчого модуля.

7.7. Оптимізація коду асемблера

7.7.1. Основи оптимізації

Наступний список містить ряд основних методів оптимізації, які часто використовуються для підвищення продуктивності x86 коду на асемблері:

- ✓ використовуйте команду `test` замість `cmp`, коли це можливо;
- ✓ уникайте використання прямої адресації в `test` і `cmp` (наприклад, `cmp dword ptr [ebp+16],100` або `test byte ptr [r12],0fh`) щоразу, коли це можливо. Завантажте значення пам'яті в регістр і використовуйте його;
- ✓ використовуйте `add` або `sub` замість `inc` або `dec`, особливо в критичних за продуктивністю петлях. Останні дві інструкції не оновлюють усі прапори стану в `EFLAGS`, що часто є досить повільним;
- ✓ використовуйте інструкції `xor`, `sub`, `pxor`, `xorps` тощо для обнулення регістру замість інструкції переміщення даних;
- ✓ уникайте використання константних 16-бітових значень в інструкціях, які вимагають префікс розміру операнда. Замість цього використовуйте еквівалентне 8-бітне або 32-бітне значення. Наприклад, використовуйте `mov edx,42` замість `mov dx,42`;
- ✓ розгорніть (або частково розгорніть) маленькі петлі, які вимагають фіксованої кількості ітерацій;
- ✓ завантажте значення з пам'яті, які необхідні для багаторазових розрахунків, у регістр. Якщо значення з пам'яті потрібне

тільки для одного розрахунку, використовуйте форму інструкції реєстр-пам'ять.

Код x86-64 можна покращити за допомогою таких методів оптимізації:

1. Використовуйте 32-бітні реєстри загального призначення і форми команд під час роботи з 32-бітними значеннями.

2. Спочатку використовуйте реєстри загального призначення EAX, EBX, ECX, EDX, ESI і EDI, а не R8D-R15D під час обробки 32-розрядних значень.

3. Використовуйте додаткові реєстри загального призначення і SIMD реєстри для того, щоб звести до мінімуму залежність даних від «викачки» реєстрів (відбувається, коли програма повинна тимчасово зберегти вміст реєстру в пам'яті, щоб звільнити реєстр для інших розрахунків).

4. Використовуйте дво- або триоперандну форму інструкції `imul`, щоб перемножити два 64-розрядних цілих числа, якщо не потрібен повний 128-бітний результат.

7.7.2. Оптимізація обчислень з плаваючою крапкою

Дотримуйтесь таких правил під час кодування функцій мовою асемблера, які використовують арифметику з плаваючою крапкою:

- ✓ використовуйте x86-SSE або x86-AVX інструкції з плаваючою крапкою замість x87 FPU в новому коді;

- ✓ уникайте втрат значущих розрядів і ненормалізованих значень під час арифметичних обчислень, коли це можливо;

- ✓ уникайте використання ненормалізованих констант з плаваючою точкою;

- ✓ якщо передбачається втрата значущих розрядів, врахуйте можливість включення режимів `flush-to-zero` (MXCSR.FZ) і `denormals-are-zero` (MXCSR.DAZ).

7.7.3. Оптимізація розгалуження програми

Інструкції розгалуження програми, такі як `jmp`, `call` та `ret`, потенційно потребують багато часу, оскільки вони можуть вплинути на зміст інтерфейсних каналів передачі і внутрішнього кешу. Команда умовного переходу `jcc` також значно впливає на продуктивність, враховуючи його частоту використання. Наступні методи оптимізації можуть бути використані для зведення до мінімуму негативних наслідків продуктивності інструкцій розгалуження і підвищити точність пророкування розгалужень:

- організація коду, щоб мінімізувати необхідні інструкції розгалуження;
- використання інструкції `setcc` і `movcc`, щоб усунути передбачувані гілки, залежні від даних;
- вирівнювання лічильника у критичних за продуктивністю петлях до межі 16 байтів;
- переміщення умовного коду, який навряд чи буде виконуватись (наприклад, код обробки помилок) в іншому розділі програми або сторінці пам'яті.

Блок пророкування розгалужень використовує як статичні, так і динамічні методи. Неправильні прогнози розгалуження можна звести до мінімуму, якщо блоки коду, що містять умовні переходи, розташовані так, що вони узгоджуються з алгоритмом статичного передбачення:

- а) використовуйте умовні переходи вперед, коли код, що йде нижче, ймовірно буде виконуватися;
- б) використовуйте зворотні умовні переходи, коли код, що йде нижче, навряд чи буде виконуватися.

Умовний перехід вперед часто використовується у блоках коду, які виконують перевірку аргументу функції. Зворотний умовний перехід може бути використано в нижній частині блоку коду циклу програми після оновлення лічильника.

7.7.4. Оптимізація роботи з даними

Важливість використання правильно вирівняних даних важко переоцінити. Програми, які працюють з неправильно вирівняними даними, найімовірніше повинні виконувати додаткові цикли пам'яті і мікрокоманди, які можуть негативно вплинути на загальну продуктивність системи. Завжди дотримуйтесь таких методів вирівнювання даних:

- ✓ вирівнюйте багатобайтні цілі значення і у форматі з плаваючою крапкою до їх природних границь;
- ✓ вирівнюйте 64, 128 і 256-бітні упаковані значення до їх власних границь;
- ✓ додавайте у структури даних резервні поля, якщо це необхідно для забезпечення правильного вирівнювання;
- ✓ використовуйте відповідні директиви компілятора і бібліотечні функції для вирівнювання елементів даних, які виділяються в коді високого рівня;
- ✓ віддавайте перевагу вирівнюванню під час зберігання щодо читання.

Також рекомендуються такі методи компонування даних:

- а) розміщення невеликих масивів і коротких текстових рядків у структурі, щоб уникнути розриву рядків кешу;
- б) оцінювання продуктивності різних макетів даних, таких як структура масивів порівняно з масивом структур.

7.7.5. Методика оптимізації SIMD-коду

Дотримуйтесь таких рекомендацій, коли використовуєте обчислювальні ресурси x86-SSE або x86-AVX:

- ✓ ліквідувати залежності регістрів для того, щоб використовувати декілька виконавчих блоків виконавчого модуля;
- ✓ використовувати упаковані значення з одинарною точністю замість значень з плаваючою крапкою подвійної точності;
- ✓ завантажувати операнди з пам'яті, які часто використовуються, в регістри;

- ✓ виконати упаковані завантаження і збереження даних за допомогою вирівняних інструкції переміщення (наприклад, *movdqa*, *movaps*, *movapd* тощо);
- ✓ обробити SIMD-масиви з використанням невеликих блоків даних для того, щоб максимізувати повторне використання даних кешу;
- ✓ використовувати перемішування замість перетасовки даних у коді x86-AVX;
- ✓ використовувати інструкцію *vzeroupper*, коли необхідно уникнути затримок у разі переходів між x86-AVX і x86-SSE;
- ✓ використовувати 32-бітну форму x86-AVX інструкції *vgather* замість 64-бітної. Виконати всі необхідні операції збору задовго до того, коли дані будуть потрібні.

Наступні методи можуть бути використані для підвищення продуктивності певних алгоритмів, які виконують операції кодування і декодування SIMD:

- ✓ використання нетимчасових команд збереження (наприклад, *movntdqa*, *movntpd*, *movntps* тощо), щоб мінімізувати забруднення кешу;
- ✓ використання інструкцій попередньої вибірки даних (наприклад, *prefetcht0*, *prefetchnta* та ін.), щоб повідомити процесор про очікуване використання елементів даних.

Контрольні питання

1. Назвіть регістри загального призначення.
2. Назвіть приклади базових типів операндів.
3. Назвіть відмінності між архітектурами x86-64 та x86-32.
4. Наведіть приклади некоректних інструкцій.
5. Назвіть характерні ознаки лінійних функцій.
6. Як відбувається робота з цілими числами в x86-64?
7. Особливості арифметики з плаваючою крапкою в асемблері x86-64.

8. Особливості роботи з масивами і рядками в асемблері x86-64.
9. Які x86 є сумісними процесорами?
10. Особливості архітектури багатоядерних процесорів.
11. Які п'ять елементарних операцій виконує ядро процесора над інструкціями?
12. У чому полягає оптимізація коду асемблера?

ТЕСТОВІ ПИТАННЯ

1. Попередником якого сучасного сімейства ОС була ОС Minix Ендрю Таненбаума:

- а) Windows;
- б) Linux;
- в) BSD.

2. Які стани задачі є загальноприйнятими й достатніми в загальному випадку:

- а) виконання;
- б) підготовка;
- в) готовність;
- г) очікування;
- д) закінчення.

3. Чи може один екземпляр мікроядерної ОС фізично бути розділеним на декілька комп'ютерів:

- а) так;
- б) ні.

4. Яка нотація викликів функції прийнята в системних викликах Windows:

- а) змішана нотація мов *C* і *Pascal* (зворотний порядок аргументів, очистка стека функцією);
- б) нотація мови *Pascal* (прямий порядок аргументів, очистка стека функцією);
- в) нотація мови *C* (зворотний порядок аргументів, очистка стека визиваючим кодом).

5. Чи можна безпосередньо звертатися до портів введення / виведення в програмах, запущених під управлінням ОС родини Windows 9x:

- а) так;
- б) ні.

6. Чи мають ядра NT+ риси мікроядерної архітектури:

- а) ні, це чистий моноліт;
- б) так, підтримка сервісів;
- в) так, підтримка сервісів і так названа «виконуюча частина» в ядрі.

7. Чи обов'язково в багатозадачній ОС буде присутня підтримка одночасного виконання декількох частин однієї і тієї самої програми:

- а) так;
- б) ні.

8. Яка з нижче наведених ОС є найбільш об'єктивно орієнтованою:

- а) усі Linux;
- б) OS/2;
- в) BeOS.

9. Як з мікроядерної архітектури називаються додатки користувача:

- а) сервери;
- б) клієнти;
- в) сервіси;
- г) програми;
- д) споживачі.

10. В якій із ОС вперше був реалізований стек протоколів TCP/IP:

- а) DOS;
- б) Linux;
- в) BSD;
- г) Windows;
- д) AmigaOS.

11. У чому основний недолік монолітних ядер:

- а) вони займають досить багато операційної пам'яті;
- б) їх не можна модифікувати під час роботи;
- в) з часом вони настільки збільшуються, що різко ускладнюється внесення будь-яких змін.

12. Укажіть типи повідомлень, які можуть використовуватись у мікроядерних ОС:

- а) синхронне й асинхронне;
- б) тільки асинхронне;
- в) тільки синхронне.

13. Які ядра сучасних ОС підтримують Multiboot Speciation:

- а) Linux;
- б) SunOS;
- в) MacOS;
- г) GNU Hurd;
- д) Windows;
- е) усі ядра BSD.

14. Які з перерахованих нижче видів архітектури ядер існують:

- а) мікроядра;
- б) екзоядра;
- в) монолітні ядра.

15. Укажіть основні функції ОС:

- а) розділення ресурсів комп'ютера між програмами;
- б) ОС повинна враховувати апаратні особливості;
- в) захист програм одна від одної і від самих себе;
- г) ОС повинна запропонувати програмам уніфікувати інтерфейс взаємодій з іншими програмами й апаратурою (через драйвери).

16. Яка оптимальна довжина кванта процесорного часу для ОС загального призначення:

- а) декілька сотень мілісекунд;
- б) декілька мілісекунд;
- в) чим менше, тим краще.

17. В яких ОС підтримка графічного інтерфейсу користувача (GUI) інтегрована безпосередньо в ядро:

- а) Windows;
- б) Linux;
- в) MacOS;
- г) BeOS;
- д) BSD.

18. Чи знайшли екзоядерні ОС широке застосування в сучасній обчислювальній техніці:

- а) ні;
- б) так.

19. Вкажіть основний засіб міжпроцесної взаємодії в мікроядерних архітектурах:

- а) віддалені виклики процедур (RPC, Remote Procedure Call);
- б) сповіщення;
- в) потоки;
- г) загальна (розділювана) пам'ять.

20. В якій операційній системі вперше було реалізовано графічний інтерфейс користувача:

- а) Windows;
- б) AmigaOS;
- в) BSD;
- г) UNIX.

21. Що означає аббревіатура PIC у контексті ОС:

- а) Portable Incompatible Code;

- б) Programmable Interrupt Controller;
- в) Position Independent Code;
- г) Past Implemented Code.

22. Які існують види багатозадачності:

- а) кругова багатозадачність;
- б) кооперативна (невитискальна) багатозадачність;
- в) витискальна багатозадачність;
- г) симетрична багатозадачність.

23. Який основний недолік мікроядерної архітектури:

- а) проблеми з переносністю;
- б) проблеми з продуктивністю;
- в) проблеми з безпекою.

24. Де розміщено драйвери щодо ядра в ОС сімейства Linux:

- а) лише в ядрі;
- б) поза ядром, лише в окремих процесах;
- в) і в ядрі, і в окремих процесах (але останнє лише у разі увімкненої підтримки в ядрі);
- г) і в ядрі, і в окремих процесах.

25. Як (в ідеалі) повинне проходити перенесення ядра ОС з однієї апаратної платформи на іншу:

- а) переписуванням будь-яких його частин;
- б) переписуванням лише однієї, виділеної частини;
- в) повним переписуванням коду.

26. Вкажіть основний засіб міжпроцесової взаємодії в мікроядерних архітектурах:

- а) дистанційні виклики процедур (RPC, Remote Procedure Call);
- б) повідомлення;
- в) потоки;
- г) спільна пам'ять.

27. Чи знайшли екзоядерні ОС широке застосування в сучасній обчислювальній техніці:

- а) ні;
- б) так.

28. Які ядра сучасних ОС підтримують Multiboot Specification:

- а) Windows;
- б) Linux;
- в) GNU Hurd;
- г) усі ядра BSD;
- д) SunOS;
- е) MacOS.

29. Вибрати всі правильні твердження про відношення DOS до перших версій Windows:

- а) у Windows можна було запускати додатки DOS;
- б) підтримка програм DOS була обмеженою і неповною (у разі емуляції на VDM, у рамках режиму V86);
- в) багато функцій Windows делегувалися відповідним функціям DOS (тобто для цього виконувалося перемикання режимів роботи ЦП).

30. Яка з нижче перерахованих ОС є найбільш об'єктно-орієнтованою:

- а) Windows 2k;
- б) OS / 2;
- в) усі Linux;
- г) BeOS;
- д) FreeBSD.

ТЕМИ ДЛЯ САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ

1. Перші операційні системи: причини виникнення і можливості.
2. Історія створення і розвитку ОС UNIX.
3. Історія створення і розвитку ОС Windows.
4. Особливості ОС Windows 10.
5. Історія створення і розвитку ОС Linux.
6. Історія створення і розвитку ОС Solaris (фірма Sun).
7. Технологія симетричної багатопроцесорності (*symmetric multiprocessing – SMP*).
8. Стани процесів: необхідність свопінгу.
9. Міжпроцесова взаємодія: базові механізми міжпроцесової взаємодії.
10. Міжпроцесова взаємодія: технології передавання повідомлень.
11. Сторінково-сегментна організація віртуальної пам'яті.
12. Реалізація керування основною пам'яттю в ОС Linux.
13. Взаємодія з диском під час керування пам'яттю: завантаження сторінок на вимогу; особливості підкачування сторінок.
14. Взаємодія з диском під час керування пам'яттю: визначення продуктивності завантаження на вимогу.
15. Огляд концепції та реалізацій хмарних обчислень.
16. Хмарна платформа Microsoft Windows Azure.
17. Огляд Google Android.
18. Аналітичний огляд і порівняння можливостей ОС для мобільних пристроїв.

СЛОВНИК ОСНОВНИХ ТЕРМІНІВ

- **Адресний простір (*Address space*)** – діапазон адрес, що доступні для комп'ютерної програми.
- **Архітектура операційної системи** – структурна і функціональна організація ОС на основі деякої сукупності програмних модулів.
- **Алгоритм Деккера** – голландський математик Т. Деккер був першим, хто розробив програмне рішення проблеми взаємного виключення (1970 р.).
- **Багатопоточність** – виконання декількох потоків в одному процесі.
- **Відкачування і підкачка (*swapping*)** – це дії операційної системи з відкачування (запису) образу неактивного процесу на диск або підкачування (зчитування) активного процесу в основну пам'ять.
- **Віртуальна пам'ять *сторінкова*** – організує переміщення даних між основною пам'яттю і диском сторінками – частинами віртуального адресного простору фіксованого і порівняно невеликого розміру.
- **Віртуальна пам'ять *сегментна*** – передбачає переміщення даних сегментами – частинами віртуального адресного простору довільного розміру.
- **Віртуальна пам'ять *сегментно-сторінкова*** – використовує дворівневий поділ, при якому віртуальний адресний простір поділяється на сегменти, а потім сегменти поділяються на сторінки. Одиницею переміщення даних є сторінка.
- **Взаємне блокування** – це блокування групи процесів, що очікують подію, яка може ніколи не статися, оскільки цю подію може викликати тільки інший процес з тієї саме групи.

Взаємоблокування зазвичай виникають під час доступу процесів до ресурсів.

- **Використання (завантаження) процесора** – відсоток часу, впродовж якого процесор зайнятий.
- **Диспетчер пам'яті (MMU – Memory Management Unit)** – відображує віртуальні адреси на фізичні адреси пам'яті.
- **Диспетчеризація без розподілу процесорного часу (багатозадачність без витіснення)** – це такий спосіб диспетчеризації, при якому активний процес виконується, доки він з власної ініціативи не передасть управління диспетчеру задач для вибору іншого, готового до виконання процесу.
- **Диспетчеризація з розподілом процесорного часу (багатозадачність з витісненням)** – це такий спосіб, де рішення про перемикання з однієї задачі на іншу приймається диспетчером задач, а не власною активною задачею.
- **Завантажувач операційної системи** – невелика програма, яка починає процес ініціалізації операційної системи після включення живлення або перезавантаження комп'ютера.
- **Кеш-пам'ять (Cache memory)** – пам'ять, що менша за розміром і швидша, ніж основна; розташована між процесором і основною пам'яттю. Кеш є буфером для комірок пам'яті, звернення до яких були останніми.
- **Критичний ресурс (КР)** – ресурс, до якого в кожний момент часу можливий доступ тільки одного процесу. Доступ до такого ресурсу здійснюється в **критичній секції (КС)** – частині коду, яка в будь-який момент часу може виконуватися тільки одним процесом.
- **Логічна адреса** – адреса, що генерована процесором під час виконання машинної команди.

- **Макроядро (*Macrokernel*)** – велике ядро операційної системи, що забезпечує широкий діапазон сервісів.

- **Мікроядро (*Microkernel*)** – мале привілейоване ядро операційної системи, що забезпечує планування процесів, управління пам'яттю; для виконання інших функцій, які традиційно пов'язують з ядром операційної системи, використовує окремі процеси.

- **Монітор** – це набір процедур, змінних та інших структур даних, об'єднаних в особливий модуль. Він являє собою високо-рівневу конструкцію мови програмування, яка забезпечує функціональність, еквівалентну до функціональності семафорів, але при цьому ним набагато легше управляти.

- **Мультипрограмування** – це такий спосіб організації обчислень, коли на однопроцесорній обчислювальній системі імітується одночасне виконання декількох програм.

- **М'ютекс** – це спрощена версія семафора: змінна, яка може перебувати в одному з двох станів – блокованому і неблокованому.

- **Операційна система (ОС)** – це програма, що керує роботою інших програм як прикладних, так і системних, і є інтерфейсом між програмами й апаратним забезпеченням комп'ютера.

- **Операційне середовище** – це інтерфейси, що необхідні програмам і користувачам для отримання сервісів ОС. Операційне середовище включає в себе **програмне середовище** та інтерфейс користувача.

- **Образ процесу (*Process image*)** – містить програму (код), дані, стек і атрибути процесу. Структура даних разом з атрибутами процесу часто називається його **керуючим блоком**.

- **Пам'ять в операційних системах** – це ресурс комп'ютера, призначений для зберігання програмного коду і даних.

- **Паралельні процеси (*concurrent*)** – їх виконання може перекриватися в часі, тобто, наприклад, другий процес починається раніше, ніж завершується перший.
- **Пастка (*Trap*)** – непрограмований умовний перехід до певної адреси, що автоматично активується апаратним забезпеченням; позиція, з якої було здійснено перехід, записується.
- **Переривання (*Interrupt*)** – призупинка нормального виконання процесу (комп'ютерної програми), що викликана зовнішньою щодо процесу подією і здійснюється так, що після її завершення виконання процесу може бути продовжено.
- **Планування** – це системний процес, який здійснює установку користувацьких процесів у чергу та визначення атрибутів їх виконання в рамках використовуваної обчислювальної системи.
- **Планування короткострокове (диспетчеризація)** – рішення про додавання процесу в пул процесів, що виконуються.
- **Планування середньострокове** – рішення про додавання процесу до кількості процесів, що повністю або частково розміщені в оперативній пам'яті.
- **Планування довгострокове** – рішення про те, який з доступних процесів буде виконуватися процесором (полягає у виборі таких обчислювальних процесів, які менше б за все конкурували між собою в процесі досягнення мети обчислень).
- **Послідовні процеси (*serial*)** – вони не є паралельними.
- **Потік** – це незалежно планований контекст виконання, що розділяє єдиний адресний простір з іншими потоками свого процесу.
- **Програмне середовище** – це системне програмне оточення, що дозволяє виконувати системні запити від прикладних програм.

- **Процес** – це окрема програма в момент її виконання, а також виділені їй ресурси комп'ютера.
- **Пропускна здатність процесора** – це кількість процесів, яка виконується процесором за одиницю часу.
- **Реєнтрабельна процедура (*Reentrant procedure*)** – підпрограма, вхід в яку може бути здійснено до завершення її попереднього виконання.
- **Ресурс** – це будь-який об'єкт, який може запитуватися й очікуватися процесом; ресурс може складатися з різної кількості ідентичних одиниць, і процес може запитувати будь-яку кількість одиниць ресурсу.
- **Розділ** – це безперервна частина фізичного диска, яку операційна система являє користувачу як окремий логічний пристрій (часто використовуються також назви «логічний диск» і «логічний розділ»).
- **Розподіл часу (*Time sharing*)** – паралельне використання пристрою декількома користувачами.
- **Розподілена операційна система** – спільна операційна система, що сумісно використовується мережею комп'ютерів. Розподілена операційна система забезпечує підтримку для міжпроцесної взаємодії, переміщення процесів, взаємовиключення, а також попередження і виявлення взаємоблокувань.
- **Семафор** – це спеціальна змінна, яка має ціле значення і пов'язану з ним чергу.
- **Симетрична багатопроцесорність (*Symmetric Multiprocessing, SMP*)** – вид багатозадачності, що дозволяє операційній системі працювати на будь-якому із доступних процесорів або на декількох доступних процесорах одночасно.

- **Система реального часу (*Real-time system*)** – операційна система, яка може планувати завдання реального часу і керувати ними.
- **Спулінг (*Spooling*)** – використання вторинної пам'яті як буфер для зниження затримок під час передавання даних між периферійними пристроями і процесорами комп'ютера.
- **Сторінка (*Page*)** – у віртуальній пам'яті блок фіксованої довжини, що має віртуальну адресу, яка передається між основною і вторинною пам'яттю.
- **Утиліти** – програми, які виконують окремі завдання управління і супроводу обчислювальної системи.
- **Файл (*file*)** – це поіменована область зовнішньої пам'яті, в яку можна записувати і з якої можна зчитувати дані (суміжна область логічного адресного простору).
- **Файлова система (ФС)** – це частина операційної системи, що включає в себе: сукупність усіх файлів на диску, набори структур даних, використовуваних для управління файлами, комплекс системних програмних засобів, що реалізують операції над файлами.
- **Файл відкачування (*backing store*)** – область дискової пам'яті, використовувана операційною системою для зберігання образів відкачаних процесів. Файл відкачування організується максимально ефективно: забезпечується прямий доступ до всіх образів процесів у пам'яті (наприклад, через таблицю за номером процесу).
- **Фізична адреса** – це реальна адреса в пам'яті, яку «бачить» і «розуміє» пристрій управління пам'яттю (*Memory Management Unit – MMU*).
- **Фізична організація файла** – це спосіб розміщення файла на диску.

- **Час обороту** – інтервал часу від моменту появи процесу у вхідній черзі до моменту завершення процесу. Цей час обороту включає в себе час очікування у вхідній черзі, час очікування у черзі готовності, час готовності у чергах до периферійних пристроїв, час виконання на процесорі та час введення / виведення.

- **Час очікування** – це сумарний час перебування процесу в черзі очікування готових процесів.

- **Час відповіді** – це час від моменту перебування процесу на вхідну чергу до моменту першого звернення процесу на введення / виведення даних.

- **Ядро (*Kernel*) операційної системи** — це набір функцій, структур даних та окремих програмних модулів, які завантажуються в пам'ять комп'ютера під час завантаження операційної системи та забезпечують три типи системних сервісів: управління введенням / виведенням інформації; управління оперативною пам'яттю; управління процесами.

- **Application Programming Interface (*API*)** – інтерфейс прикладного програмування.

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

- **FAT (*file allocation table*)** – таблиця розміщення файлів.

- **Instruction Set Architecture** – архітектура системи команд.

- **Hardware Abstraction Layer (*HAL*)** – рівень апаратних абстракцій.

- **Page fault (*переривання через відсутність сторінки*)** – відбувається тоді, коли сторінка, що містить слово, до якого здійснюється звернення, відсутня в основній пам'яті. Це викликає

генерацію переривання, обробник якого завантажує необхідну сторінку в основну пам'ять.

- **Symmetric Multiprocessing, SMP** – симетрична багато-процесорність.

- **Swapping (*відкачування і підкачка*)** – це дії операційної системи з відкачування (запису) образу неактивного процесу на диск або підкачування (зчитування) активного процесу в основну пам'ять. Потреба у виконанні подібних дій викликана нестачею основної пам'яті.

- **Job control language, JCL (*мова керування завданнями*)** – мова, що розроблена для опису завдань та їх вимог до операційної системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. William Stallings. Operating Systems: Internals and Design Principles, 9th Edition. – Pearson, 2018. ISBN-10: 0-13-467095-7 | ISBN-13: 978-0-13-467095-9.
2. Stallings, William. Operating systems: internals and design principles / William Stallings. – 7th ed. Prentice Hall, New Jersey, 2012, p. 769. ISBN-13:978-0-13-230998-1
3. Kusswurm Daniel. Modern X86 Assembly Language Programming / Daniel Kusswurm. – Apress, 2019. – 604 p.
4. Бэкон Дж. Операционные системы / Дж. Бэкон, Т. Харрис. – К.: Издат. группа BHV, 2004. – 800 с.
5. Дейтел П. Дж. Операционные системы. Часть 1. Основы и принципы / П. Дж. Дейтел, Х. М. Дейтел, Д. Р. Чофнес. – 3-е изд.; пер. с англ. – Бином-Пресс, 2011. – 1024 с.
6. Митчелл М. Программирование для Linux. Профессиональный подход / М. Мітчелл, Д. Оулдем, А. Самьюэл. – Вильямс, 2002. – 288 с.
7. Стивенс У. Р. UNIX: взаимодействие процессов / У. Р. Стивенс. – СПб.: Питер, 2003. – 576 с.
8. Стивенс У. Р. UNIX: разработка сетевых приложений / У. Р. Стивенс, Б. Феннер, Э. М. Рудофф. – 3-е изд. – 2007. – 1 040 с.
9. Столлинс В. Операционные системы / В. Столлинс. – 4-е издание; пер. с англ. – М.: Издательский дом «Вильямс», 2002. – 848 с.
10. Таненбаум Э. Операционные системы / Э. Таненбаум. – 3-е изд. – СПб.: Питер, 2010. – 1 120 с.
11. Фуско Дж. Linux. Руководство программиста / Дж. Фуско. – СПб.: Питер, 2010. – 448 с.
12. Харт Дж. В. Системное программирование в среде Win32 / Дж. В. Харт. – М.: Вильямс, 2001. – 464 с.
13. Haseman Chris. Android Essentials / Chris Haseman. – Apress, 2008. – 116 p.

14. Уорд Б. Внутреннее устройство Linux / Б. Уорд, 2016. – 864 с.
15. Шоттс У. Командная строка Linux. Полное руководство / У. Шоттс. – СПб., 2017. – 480 с.
16. Дэйв Марк и др. iOS 6 SDK. Разработка приложений для iPhone, iPad и iPod touch = Beginning iOS 6 Development Exploring the iOS SDK / Дэйв Марк и др. – М.: Вильямс, 2013. – 672 с. – ISBN 978-5-8459-1852-9.
17. Sebesta, Robert W. Concepts of programming languages / Robert W. Sebesta. – 10th ed. p. cm. Includes bibliographical references and index. ISBN 978-0-13-139531-2 (alk. paper) 1. Programming languages (Electronic computers) I. Title. QA76.7.S43 2009 005.13—dc22
18. Кнут Д. Э. Искусство программирования / Д. Э. Кнут. – 3-е изд. – М.: Вильямс, 2015.
19. Себеста Роберт У. Основные концепции языков программирования / У. Себеста Роберт. – 5-е изд.; пер. с англ. – М.: Вильямс, 2001. – 672 с.
20. Васильев А. Н. Самоучитель C++ с задачами и примерами: учебное пособие / А. Н. Васильев. – СПб.: Наука и Техника, 2010. – 480 с.: илл.
21. Рассел С. Искусственный интеллект. Современный подход / С. Рассел, П. Норвиг. – 2-е изд. – М.: Вильямс, 2007. – 1 410 с. – ISBN 5-8459-0887-2, 0-13-790395-2, 978-5-8459-0887-2.

Навчальне видання

Федотова-Півень Ірина Миколаївна,
Півень Олег Борисович

ОПЕРАЦІЙНІ СИСТЕМИ

Навчальний посібник

Відповідальний за випуск	<i>А. В. Лавренюк</i>
Відповідальний редактор	<i>І. Є. Врублевська</i>
Редактор	<i>М. М. Грабарчук</i>
Форматування та комп'ютерна верстка	<i>О. В. Лисенко</i>

Здано до друку 30.01.2022. Формат 60 × 84 / 16
Папір офсетний № 1. Гарнітура «Times New Roman»
Друк. арк. 16,5
Тираж 300 примірників. Замовлення № 1 012

Підготовлено до друку Видавничо-поліграфічним центром
Університету ДФС України
08205, вул. Університетська, 31, м. Ірпінь, Київська область, Україна

*Свідоцтво про внесення суб'єкта видавничої справи
до державного реєстру видавців, виготовлювачів і
розповсюджувачів видавничої продукції
Серія ДК № 5104 від 20.05.2016*