МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**ЧЕРКАСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ БОГДАНА ХМЕЛЬНИЦЬКОГО**

Авраменко В. С., Авраменко А. С.

ОСНОВИ ОПЕРАЦІЙНИХ СИСТЕМ

Навчальний посібник

Черкаси 2018

УДК 004.451

Рецензенти: *В. М. Рудницький,* доктор технічних наук, професор, завідувач кафедри інформаційної безпеки та комп'ютерної інженерії Черкаського державного технологічного університету;
В. І. Салапатов, кандидат технічних наук, доцент кафедри програмного забезпечення автоматизованих систем Черкаського національного університету імені Богдана Хмельницького.

Рекомендовано до друку рішенням Вченої ради
Черкаського національного університету імені Богдана Хмельницького
(Протокол № 2 від 17.10.2017 р.)

Авраменко В. С., Авраменко А. С.
Основи операційних систем. Навчальний посібник. – Черкаси: ЧНУ імені Богдана Хмельницького, 2018. – 524 с.: іл.
ISBN 966-552-157-8

У посібнику розглянуті основні питання дисципліни «Операційні системи»: огляд операційних систем, проектування та архітектури операційних систем, управління процесами і потоками, мультіпроцессорна обробка, методи синхронізації процесів, проблема тупиків, методи управління пам'яттю, основи організації введення-виведення і файлових систем.

Навчальний посібник призначений для студентів ВНЗ, які навчаються за спеціальностями «121 Інженерія програмного забезпечення», «122 Комп'ютерні науки та інформаційні технології», «124 Системний аналіз» та для студентів інших споріднених спеціальностей.

УДК 004.451

ISBN 966-552-157-8

© Авраменко В. С.
© Авраменко А. С.

ЗМІСТ

ВСТУП ... 12
1 ІСТОРІЯ РОЗВИТКУ ОПЕРАЦІЙНИХ СИСТЕМ ... 14
1.1 Ранні обчислювальні пристрої ... 14
1.2 Електромеханічні і релейні машини ... 15
1.3 Послідовна обробка даних ... 16
1.4 Прості пакетні системи ... 18
1.5 Багатозадачні пакетні системи ... 19
1.6 Системи з режимом розподілу часу ... 22
1.7 Операційні системи і глобальні мережі ... 25
1.8 Операційні системи міні-комп'ютерів ... 26
1.9 Основні досягнення ... 27
1.10 Сучасні системи UNIX ... 31
1.11 Маленький UNIX виростає в LINUX ... 31
1.11.1 Історія створення Linux ... 32
1.11.2 Системні характеристики Linux ... 32
1.11.3 Відмінності між Linux і іншими ОС ... 34
1.11.4 Розвиток системи Linux ... 34
1.12 Операційна система OS/2 ... 35
1.13 Нові досягнення Microsoft Windows ... 38
1.14 Історія операційних систем Apple ... 39
1.15 Операційні системи для мобільних пристроїв ... 50
1.15.1 Історія ОС Google Android ... 50
1.15.2 Історія ОС Windows Phone ... 52
1.16 ОС для хмарних обчислень ... 54
1.17 Операційні системи СРСР ... 55
Контрольні питання і тести до розділу 1 ... 56
2 ПРИЗНАЧЕННЯ І КЛАСИФІКАЦІЯ ОС ... 59
2.1 Різноманітність операційних систем ... 60
2.2 Призначення і функції операційної системи ... 63
2.2.1 Основні функції операційних систем ... 63
2.2.2 ОС як віртуальна машина ... 65
2.2.3 ОС як диспетчер ресурсів ... 66
2.2.4 Управління процесами ... 67
2.2.5 Управління пам'яттю ... 69
2.2.6 Управління файлами і зовнішніми пристроями ... 70
2.2.7 Захист даних і адміністрування ... 71
2.2.8 Інтерфейс прикладного програмування ... 72
2.2.9 Інтерфейс користувача ... 73

2.3 Можливості розвитку ОС 73
2.4 Компоненти комп'ютерної системи 74
2.5 Класифікація ОС 74
2.5.1 Особливості алгоритмів управління ресурсами 74
2.5.2 Особливості апаратних платформ 76
2.5.3 Особливості областей використання 76
2.5.4 Особливості методів побудови 77
Контрольні питання і тести до розділу 2 78
3 СУЧАСНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ОС 80
3.1 Вимоги, що пред'являються до операційної системи 80
3.2 Розширюваність 81
3.3 Переносимість 82
3.4 Сумісність 83
3.5 Надійність, захист інформації і безпека 84
3.6 Структура побудови ОС. Архітектура ядра 86
3.6.1 Ядро і допоміжні модулі ОС 86
3.6.2 Ядро в привілейованому режимі 88
3.6.3 Монолітні системи 90
3.6.4 Багаторівневі (багатошарові) системи 92
3.6.5 Модель клієнт-сервер і мікроядро 97
3.6.6 Об'єктно-орієнтований підхід 104
3.7 Множинні прикладні середовища 105
3.8 Концепція віртуальних машин 106
3.9 Мережеві і розподілені ОС 106
3.10 Мультипрограмування 107
3.10.1 Мультипрограмування в системах пакетної обробки 108
3.10.2 Мультипрограмування в системах розподілу часу 111
3.10.3 Мультипрограмування в системах реального часу 112
3.11 Багатопоточність 113
3.12 Симетрична багатопроцесорна обробка 113
3.12.1 Архітектура симетричної багатопроцесорності 114
3.12.2 Організація симетричної багатопроцесорної системи 118
3.12.3 Архітектура багатопроцесорних ОС 118
Контрольні питання і тести до розділу 3 121
4 АПАРАТНА ПІДТРИМКА РОБОТИ ОС 125
4.1 Типові засоби апаратної підтримки ОС 125
4.1.1 Засоби підтримки привілейованого режиму 125
4.1.2 Засоби трансляції адрес 126

4.1.3 Засоби перемикання процесів 126
4.1.4 Система переривань 126
4.1.5 Системний таймер 128
4.1.6 Засоби захисту областей пам'яті 128
4.2 Машинно-залежні компоненти ОС 128
4.3 Основні елементи комп'ютера 130
4.4 Процесори 130
4.4.1 Регістри процесора 130
4.4.2 Виконання команд 135
4.5 Кешування даних 140
4.5.1 Ієрархія запам'ятовуючих пристроїв 140
4.5.2 Кеш-пам'ять 141
4.5.3 Принцип дії кеш-пам'яті 142
4.5.4 Проблема узгодження даних 144
4.5.5 Способи відображення основної пам'яті на кеш 145
4.6 Засоби підтримки механізмів віртуальної пам'яті 148
4.6.1 Віртуальний адресний простір 148
4.6.2 Перетворення адрес 151
4.6.3 Сегментно-сторінковий механізм 153
4.7 Кешування в процесорі Pentium 157
4.7.1 Буфер асоціативної трансляції 158
4.7.2 Кеш першого рівня 161
4.8 Шини 162
4.9 Технології введення-виведення 164
Контрольні питання і тести до розділу 4 165
5 УПРАВЛІННЯ ПРОЦЕСАМИ 167
5.1 Поняття процесу 167
5.2 Поняття ресурсу 170
5.3 Модель процесу 173
5.4 Створення процесу 175
5.5 Завершення процесу 176
5.6 Стани процесів 177
5.7 Призупинені процеси 179
5.8 Опис процесів 183
5.8.1 Управляючі структури ОС 184
5.8.2 Структури управління процесами 185
5.8.3 Атрибути процесів 187
5.8.4 Управління процесами 188

5.9 Операції над процесами ... 190
5.9.1 Набір операцій ... 190
5.9.2 Одноразові операції ... 190
5.9.3 Багаторазові операції ... 191
5.9.4 Перемикання контексту ... 192
5.10 Взаємодія процесів ... 193
5.10.1 Пам'ять, що розділяється ... 194
5.10.2 Сигнали ... 194
5.10.3 Передача повідомлень ... 195
5.10.4 Віддалений виклик процедур ... 196
5.10.5 Сокети ... 197
Контрольні питання і тести до розділу 5 ... 197
6 УПРАВЛІННЯ ПОТОКАМИ ... 200
6.1 Концепції потоку ... 200
6.2 Модель потоку ... 204
6.3 Використання потоків ... 206
6.4 Стани потоку ... 208
6.5 Потоки на рівні користувача і на рівні ядра ... 210
6.5.1 Потоки на рівні користувача (ULT) ... 210
6.5.2 Потоки на рівні ядра (KLT) ... 213
6.5.3 Комбіновані підходи ... 214
6.5.4 Спливаючі потоки ... 214
6.6 Багатопотокове програмування ... 214
6.7 Потоки у Windows 2000 ... 216
6.7.1 Багатопоточність ... 217
6.7.2 Стани потоків в ОС Windows ... 220
6.7.3 Підтримка симетричної багатопроцесорної обробки ... 221
6.8 Управління процесами і потоками в LINUX ... 222
6.8.1 Процеси в LINUX ... 222
6.8.2 Потоки в LINUX ... 224
Контрольні питання і тести до розділу 6 ... 226
7 ВЗАЄМНІ ВИКЛЮЧЕННЯ І БАГАТОЗАДАЧНІСТЬ ... 231
7.1 Принципи паралельних обчислень ... 231
7.1.1 Участь операційної системи ... 231
7.1.2 Взаємодія процесів ... 232
7.1.3 Вимоги до взаємних виключень ... 235
7.2 Взаємне виключення через загальні змінні ... 236

7.3 Взаємні виключення з використанням системних функцій 238
7.4 Взаємовиключення: програмний підхід 239
7.4.1 Алгоритм Деккера 239
7.4.2 Алгоритм Петерсона 240
7.4.3 Алгоритм Лемпорта 241
7.5 Взаємне виключення: апаратна підтримка 242
7.5.1 Команда TestAndSet 243
7.5.2 Команда обміну 244
Контрольні питання і тести до розділу 7 245
8 СЕМАФОРИ, МОНІТОРИ І ПОВІДОМЛЕННЯ 247
8.1 Семафори 247
8.1.1 Взаємні виключення 250
8.1.2 Задача «Виробник-Споживач» («Письменник-Читач») 251
8.1.3 Java семафори 253
8.2 Монітори 258
8.2.1 Монітори з сигналами (монітори Хоара) 259
8.2.2 Монітори із сповіщенням і широкомовленням 263
8.3 Передача повідомлень 267
8.3.1 Примітиви передачі повідомлень 267
8.3.2 Синхронізація 268
8.3.3 Адресація 269
8.3.4 Формат повідомлення 270
8.3.5. Взаємні виключення 270
Контрольні питання і тести до розділу 8 272
9 ВЗАЄМНЕ БЛОКУВАННЯ 275
9.1 Принципи взаємного блокування 275
9.2 Умови виникнення тупиків 278
9.3 Моделювання взаємного блокування 278
9.4 Ігнорування проблеми взаємних блокувань 280
9.5 Виявлення взаємних блокувань 280
9.5.1 Використання одного ресурсу кожного типу 280
9.5.2 Використання декількох ресурсів кожного типу 283
9.5.3 Безпечний і небезпечний стан 286
9.5.4 Реалізація алгоритму виявлення тупика 288
9.6 Відновлення після взаємного блокування 289
9.6.1 Відновлення за допомогою перерозподілу ресурсів 289
9.6.2 Відновлення шляхом відкату 290

9.6.3 Відновлення шляхом знищення процесів 290
9.7 Ухилення від взаємних блокувань 291
9.7.1 Заборона запуску процесу 291
9.7.2 Алгоритм банкіра для одного ресурсу 292
9.7.3 Алгоритм банкіра для декількох типів ресурсів 293
9.7.4 Реалізація алгоритму банкіра 295
9.7.5 Недоліки алгоритму банкіра 297
9.8 Запобігання взаємних блокувань 298
9.8.1 Порушення умови взаємних блокувань 298
9.8.2 Порушення умови утримання і очікування ресурсів 299
9.8.3 Порушення принципу відсутності перерозподілу 299
9.8.4 Порушення умови кругового очікування 299
9.9 Зависання 301
9.10 Тупики в системах спулінгу 301
Контрольні питання і тести до розділу 9 302
10 УПРАВЛІННЯ ПАМ'ЯТТЮ 306
10.1 Основні поняття і вимоги до управління пам'яттю 306
10.1.1 Переміщення 306
10.1.2 Захист 306
10.1.3 Спільне використання 307
10.1.4 Типи адрес 307
10.2. Розподіл пам'яті фіксованими розділами 309
10.3 Динамічний розподіл пам'яті 311
10.4 Переміщувані розділи 313
10.5 Система двійників 315
10.6 Поняття віртуальної пам'яті 316
10.7 Сторінковий розподіл пам'яті 318
10.8 Багаторівневі таблиці сторінок 324
10.9 Буфери швидкого перетворення адреси (TLB) 327
10.10 Інвертовані таблиці сторінок, хеш-таблиці 331
10.11 Визначення найкращого розміру сторінки 333
10.12 Сегментна організація віртуальної пам'яті 334
10.13 Сегментно-сторінкова організація пам'яті 340
10.13.1 Загальна схема організації сегментно-сторінкової пам'яті 341
10.13.2 Сегментно-сторінкова організація пам'яті в Windows 343
10.14 Свопінг 350
Контрольні питання і тести до розділу 10 351
11 УПРАВЛІННЯ СТОРІНКОВОЮ ПАМ'ЯТТЮ 357

11.1 Стратегії управління сторінковою пам'яттю 357
11.2 Основні алгоритми заміщення сторінок 358
11.3 Оптимальний алгоритм 360
11.3.1 Першим увійшов – першим вийшов 360
11.3.2 Годинниковий алгоритм 361
11.3.3 Заміщення сторінки, яка найдовше не використовувлася 364
11.3.4 Буферизація сторінок 367
11.3.5 Стратегія заміщення і розмір кеша 368
Контрольні питання і тести до розділу 11 369
12 ПЛАНУВАННЯ ПРОЦЕСІВ 371
12.1 Планування в системах з одним процесором 371
12.1.1 Рівні планування процесів 372
12.1.2 Параметри планування 374
12.1.3 Алгоритми планування 376
12.2 Алгоритми планування, засновані на квантуванні 379
12.3 Алгоритми планування, засновані на пріоритетах 381
12.4 Змішані алгоритми планування 384
12.5 Альтернативні стратегії планування 386
12.6 Планування в Windows 2000 394
12.6.1 Процеси і потоки 394
12.6.2 Планування процесів і потоків 396
12.6.3 Синхронізація потоків 401
12.6.4 Взаємодія між процесами 402
Контрольні питання і тести до розділу 12 404
13 БАГАТОПРОЦЕСОРНІ СИСТЕМИ 412
13.1 Історія багатопроцесорної обробки 412
13.2 Форми паралелізму 415
13.3 Класифікація обчислювальних систем 416
13.4 Варіанти об’єднання мультипроцесорних систем 421
13.5 Мультипроцесор із загальною пам’яттю 424
13.5.1 Мультипроцесорне апаратне забезпечення 424
13.5.2 Мультипроцесорні ОС 426
13.5.3 Планування роботи мультипроцесора 429
13.6 Мультипроцесорні системи з передачею повідомлень 434
Контрольні питання і тести до розділу 13 437
14 ПЛАНУВАННЯ РЕАЛЬНОГО ЧАСУ 439
14.1 Поняття систем реального часу 439

14.2 Характеристики ОС реального часу ... 440
14.3 Планування реального часу ... 441

Контрольні питання і тести до розділу 14 ... 445

15 УПРАВЛІННЯ ВВЕДЕННЯМ-ВИВЕДЕННЯМ ... 447

15.1 Фізична організація пристроїв введення-виведення ... 447
15.2 Задачі ОС з управління файлами і пристроями ... 448

15.2.1 Організація паралельної роботи пристроїв уведення-виведення ... 448
15.2.2 Узгодження швидкостей обміну і кешування даних ... 449
15.2.3 Розподіл пристроїв і даних між процесами ... 450
15.2.4 Забезпечення логічного інтерфейсу між пристроями ... 451
15.2.5 Підтримка широкого спектру драйверів ... 452
15.2.6 Динамічне завантаження і вивантаження драйверів ... 453
15.2.7 Підтримка декількох файлових систем ... 454
15.2.8 Синхронні і асинхронні операції введення-виведення ... 454

15.3 Організація програмного забезпечення введення-виведення ... 455
15.4 Буферизація операцій введення-виведення ... 459
15.6 Дискове планування ... 463

15.6.1 Будова жорсткого диска і параметри планування ... 463
15.6.2 Параметри продуктивності диска ... 464
15.6.3 Стратегії дискового планування ... 465

15.7 Диспетчеризація дискових операцій ... 465

15.7.1 Алгоритм First Come First Served ... 466
15.7.2 Алгоритм диспетчеризації SSTF ... 466
15.7.3 Алгоритм диспетчеризації SCAN ... 467
15.7.4 Алгоритм диспетчеризації C-SCAN ... 468

Контрольні питання і тести до розділу 15 ... 469

16 ФАЙЛОВА СИСТЕМА ... 474

16.1 Файли ... 474

16.1.1 Імена файлів ... 474
16.1.2 Типи файлів ... 475
16.1.3 Атрибути файлів ... 475

16.2 Логічна організація файлу ... 476
16.3 Фізична організація файлу ... 479

16.3.1 Диски, розділи, сектори, кластери ... 480
16.3.2 Фізична організація і адресація файлу ... 482

16.4 Файлова система FAT ... 485

16.4.1 Таблиця розміщення файлів ... 487

16.4.2 Файлова система FAT32 489
16.5 Файлова система NTFS 491
16.5.1 Ключові можливості NTFS 492
16.5.2 Том NTFS і файлова структура 492
16.5.3 Схема тома NTFS 493
16.5.4 Головна файлова таблиця 493
16.5.5 Цілісність даних і відновлення в NTFS 498
16.5.6 Компресія файлів і каталогів 498
16.6 Файлова система ReFS 499
16.6.1 Особливості файлової системи ReFS 499
10.6.2 Відмінності файлової системи ReFS від NTFS 501
16.7 Файлова система Mac OS 501
Контрольні питання і тести до розділу 16 503
17. ВІДПОВІДІ НА КОНТРОЛЬНІ ПИТАННЯ І ТЕСТИ 506
17.1 Відповіді на контрольні питання 506
17.2 Відповіді на тести 513
СПИСОК ЛІТЕРАТУРИ 522

ВСТУП

У навчальному посібнику розглядаються питання побудови і функціонування не конкретної сучасної операційної системи (ОС) і навіть не йде мова про конкретний тип операційних систем. Операційні системи розглядаються з найзагальніших позицій, а описувані фундаментальні концепції і принципи побудови справедливі для більшості ОС. У переважній більшості ОС зустрічаються одні і ті ж принципи управління ресурсами комп'ютера: мультипрограмування і мультипроцесування, віртуальна пам'ять і свопінг, файли, що відображаються в пам'яті, і віддалений виклик процедур. Тому в навчальному посібнику ми не прив'язуємося до якої-небудь конкретної ОС. Тільки невелика частина посібника присвячена тому, як розглянуті принципи реалізовані в конкретних сучасних системах.

Попередниками сучасних ОС були системи пакетної обробки і однозадачні операційні системи. Але розвиток комп'ютерних технологій сформував необхідність реалізації нових можливостей, серед яких дуже важливими були розподіл часу, багатозадачність, розрахована на багато користувачів і мережевий режими роботи. Своє повне втілення названі принципи отримали в сімействах таких ОС як Windows, OS/2, UNIX та інших сучасних ОС. Ці операційні системи з перших версій підтримували вказані технології.

Пропонований увазі читачів навчальний посібник написаний за матеріалами курсу «Операційні системи», який читається студентам спеціальностей «121 Інженерія програмного забезпечення», «122 Комп' ютерні науки та інформаційні технології», «124 Системний аналіз» Черкаського національного університету ім. Богдана Хмельницького.

Перші три розділи вводять читача в проблематику операційних систем. Після розгляду основних етапів еволюції ОС розглядається призначення сучасної операційної системи. Потім описується функціональна і структурна організація ОС, а також основні підсистеми і компоненти, які використовуються для управління як локальними, так і такими, що розділяються, мережевими ресурсами. При цьому розглядається класична багатошарова організація ОС з монолітним ядром, а також мікроядерна архітектура.

У четвертому розділі розглядаються засоби апаратної підтримки ОС. Багато операційних систем успішно працюють на різних апаратних платформах без істотних змін у своєму складі. Багато в чому це пояснюється тим, що, незважаючи на відмінності в деталях, засоби апаратної підтримки ОС більшості комп'ютерів набули сьогодні багато типових рис. У результаті в ОС можна виділити досить компактний шар машинно-залежних компонентів ядра і зробити інші шари ОС загальними для різних апаратних платформ.

Наступні чотири розділи (з п'ятого по восьмий) присвячені концепціям і механізмам управління локальними ресурсами комп'ютера: процесором, пам'яттю і зовнішніми пристроями. Вивчаються поняття процесу і потоку, різні механізми управління процесами і потоками, вживані в системах пакетної обробки, розподілу часу і реального часу. Досить детально розглядаються різні

алгоритми взаємних виключень і механізми синхронізації вищого рівня – семафори і монітори.

У дев'ятому розділі розглядається проблема тупиків і наводяться основні результати наукових досліджень, присвячених питанням запобігання, обходу, виявлення тупиків і питанням відновлення після них. Розглядається також тісно пов'язана з тупиками проблема нескінченного відкладання.

Десятий і одинадцятий розділи присвячені методам управління пам'яттю в історичній ретроспективі – від найпростіших схем з фіксованими розділами до сучасної сторінкової і сегментно-сторінкової організації пам'яті. Детально досліджуються сучасні алгоритми заміщення сторінок.

У наступних трьох розділах (12, 13 і 14) розглядаються різні рівні і алгоритми планування процесів в системах з одним і декількома процесорами.

У останніх двох розділах (15, 16) розглядається одне з головних завдань ОС – забезпечення обміну даними між додатками і периферійними пристроями комп'ютера. У сучасній ОС функції обміну даними з периферійними пристроями виконує підсистема введення-виведення і файлова система. Як приклади використовуються найпоширеніши файлові системи, такі як NTFS і FAT. Вивчаються такі важливі функції файлових систем, як стійкість до збоїв і відмов, а також контроль доступу до даних, що зберігаються.

У описі багатьох алгоритмів і даних у даному посібнику ми орієнтувалися на мову програмування С або Псевдо-С, оскільки в деяких прикладах повністю не дотримуються синтаксичні правила мови там, де строге наслідування їх вело, на нашу думку, до зайвої конкретизації.

1 ІСТОРІЯ РОЗВИТКУ ОПЕРАЦІЙНИХ СИСТЕМ

Серед усіх системних програм, з якими доводиться мати справу користувачам комп'ютерів, особливе місце займають операційні системи. Операційна система (ОС) – це програма, яка запускається відразу після включення комп'ютера і дозволяє користувачеві управляти комп'ютером**.**

Операційна система управляє комп'ютером, запускає програми, забезпечує захист даних, виконує різні сервісні функції за запитами користувача і програм. Кожна програма користується послугами ОС, а тому може працювати тільки під управлінням тієї ОС, яка забезпечує для неї ці послуги.

Історія розвитку операційних систем тісно пов'язана з історією розвитку комп'ютерів, оскільки ОС з'явилися і розвивалися в процесі конструювання комп'ютерів. Тому, щоб представити як виглядали ОС, ми коротко і послідовно обговоримо покоління деяких моделей комп'ютерів, для яких, в основному, створювалися ОС [5].

1.1 Ранні обчислювальні пристрої

В якості першої в історії обчислювальної машини називають механічний арифмометр Вільгельма Шиккарда, створений в 1623 році. Машина була названа «рахунковими годинником», оскільки була заснована на механічних деталях, характерних для годинника. «Рахунковий годинник» оперував шестирозрядними цілими числами і здатний був робити додавання і віднімання. Переповнювання відзначалося дзвоном дзвоника. До наших днів машина не збереглася, але в 1960 році була створена працююча копія [5].

Найстарішою з рахункових машин, що збереглися до наших днів, є арифмометр Блеза Паскаля, створений в 1645 році. Паскаль почав роботу над машиною в 1642 році у віці 19 років. Батько винахідника працював збирачем податків і змушений був проводити довгі виснажливі підрахунки. Своїм винаходом Блез Паскаль сподівався полегшити роботу батька.

Перший зразок мав п'ять десяткових дисків, тобто міг працювати з п'ятизначними числами. Пізніше були створені машини, що мали до восьми дисків. Додавання на машині Паскаля виконувалося легко, що ж до віднімання, то для нього доводилося використати метод дев'ятичних доповнень [5].

Через тридцять років Лейбніц побудував механічну машину, здатну виконувати додавання, віднімання, множення і ділення, а також обчислення квадратного кореня. Окрім цього, саме Лейбніц запропонував двійкову систему представлення чисел, яка зараз використовується в усіх обчислювальних машинах.

У 1820 році Чарльз Томас створив машину, названу просто «***арифмометр»***. Арифмометр мав пристрій, схожий з машиною Лейбніца, і виконував ті ж операції. Робота Томаса цікава в основиому тим, що саме його арифмометр став першою рахунковою машиною, запущеною в серійне виробництво [5].

Англійський математик Чарлз Беббідж (1792-1871) в 1823 році почав роботу над складнішою машиною – ***різницевою***. Ця машина повинна була

реалізовувати метод кінцевих різниць для побудови математичних таблиць. Робота частково фінансувалася англійським урядом. Спочатку Беббідж планував побудувати машину, яка працювала з шестирозрядними числами і обчислювала різниці другого порядку.

У 1830 році в результаті конфлікту Беббіджа з інженером Йозефом Клементом, найнятим раніше для роботи над машиною, створення машини призупинилося. Самого Беббіджа це не збентежило. Тепер він планував створення машини, яка працювала з двадцятирозрядними числами з використанням різниці шостого порядку. Більше того, в 1834 році Беббідж зовсім втратив інтерес до різницевої машини, дійшовши до висновку, що будувати слід машину універсальну, не обмежену у своїй роботі одним завданням. Цю машину він назвав ***аналітичною***. Це була чисто механічна машина, але Беббідж розумів, що для аналітичної машини йому необхідно програмне забезпечення.

На жаль, аналітична машина так і не була побудована. Роботи над різницевими машинами обійшлися англійському уряду в 17000 фунтів стерлінгів (таку ж кількість грошей Беббідж витратив зі свого бюджету). Не отримавши ніякої працюючої машини, уряд відмовився фінансувати подальші дослідження Беббіджа. Беббідж встиг виконати повні креслення майбутньої машини і навіть утілити «в металі» деякі її вузли. Попри те, що машина так і не була побудована, саме вона стала першою спробою створити ***програмовану обчислювальну машину***.

Подорожуючи по Італії Беббідж познайомився з італійським математиком Луїджі Меиабрі, який у 1842 році опублікував французькою мовою статтю з описом машини Беббіджа. Статтю переклала на англійську в 1843 році леді Ада Аугуста Лавлейс, дочка поета Байрона. Леді Лавлейс забезпечила свій переклад розгорнутими коментарями, що значно переважали за розміром саму статтю. В одному з розділів цих коментарів наводиться повний набір команд для обчислення на аналітичній машині чисел Бернуллі. Цей набір команд вважається першою в історії комп'ютерною програмою, а сама Ада Лавлейс – першим програмістом. В її честь названа мова програмування Ada [5].

1.2 Електромеханічні і релейні машини

У кінці 1930-х років німецький інженер Конрад Цузе почав роботу над електромеханічними обчислювальними машинами. Перша машина, яка була створена в 1937 році і названа Zl, яка сприймала інструкції з кінострічки (перфострічки), була електромеханічним калькулятором з обмеженими можливостями програмування.

Наступна машина Конрада Цузе, Z2, була заснована на телефонних реле. У 1941 році Цузе завершив роботу над машиною Z3. що була першим повністю функціонуючим програмованим обчислювальним пристроєм. Машина мала 2200 реле, працювала з тактовою частотою 5-10 Гц і мала довжину машинного слова 22 біти. Z3 використовувала двійкову арифметику.

Ідею використання реле для реалізації двійкової арифметики приписують Клоду Шеннону, що запропонував відображення булевої алгебри на

електромагнітні реле у своїй магістерській дисертації в 1937 році. Так або інакше, Конрад Цузе уперше успішно застосував двійкову арифметику в реально працюючій машині.

Оригінал Z3 був знищений в 1944 році при бомбардуваннях Берліна авіацією союзників. Знищені були приміщення заснованої Цузе компанії Zuse Apparatebau. На щастя, майже завершена на той час машина Z4 була раніше евакуйована в безпечне місце. У 1960 році машина Z3 була відтворена в якості експоната для Німецького технічного музею.

У 1950 році завершена машина Z4 стала першим у світі комп'ютером, проданим за гроші.

Якщо Аду Лавлейс слід вважати першим програмістом-теоретиком, то Конрад Цузе, мабуть, є першим програмістом-практиком.

Друга світова війна перешкодила роботам Цузе зробити серйозний вплив на світові наукові розробки в області автоматизації обчислень, що, проте, не зменшує його заслуги як творця першої працюючої програмованої обчислювальної машини.

Тим часом в США, в Bell Labs Джордж Стібітс також розробляв обчислювальні машини на основі реле. Перша працююча машина була створена в 1938 році. У 1940 році Стібітс продемонстрував машину, що виконувала обчислення над комплексними числами. Ця розробка відома також як перша машина, управляти якою можна було віддалено по телефонній лінії за допомогою телетайпу. Машина була продемонстрована на конференції, серед учасників якої були Джон Фон Нейман, Джои Моушлі і Норберт Вінер.

1.3 Послідовна обробка даних

Перше покоління комп'ютерів (1943-1955): механічні реле, електронні лампи і комутаційні панелі. У 1938 році Вінсент Атанасов і Кліффорд Беррі (університет штату Айова) створили спеціалізовану машину для розв'язання систем лінійних рівнянь, уперше застосувавши радіолампи. У певному значенні саме їх комп'ютер, названий ABC (Atanasoff Berry Computer), став першою в історії ***електронно-обчислювальною машиною***.

Друга світова війна також зіграла певну роль в розвитку обчислювальної техніки. Так, у Великобританії за участю Алана Т'юринга була створена повністю електронна машина Colossus (1943), що призначалася для розшифровки перехоплених німецьких повідомлень. Colossus розроблялася і експлуатувалася в обстановці строгої секретності; подробиці проекту стали доступні громадськості тільки через 30 років.

У певному значенні більше повезло американському проекту ENIAC, виконаному в університеті штату Пенсильванія Джоном Преспером Еккертом і Джоном Уільямом Моушлі. Створення ENIAC фінансувалося з військового бюджету США і мало на меті автоматизацію розрахунку таблиць наведення важкої артилерії. Роботи над машиною були завершені тільки в 1946 році, коли війна вже закінчилася. Можливо, саме цим обумовлена доступність інформації про проект ENIAC для наукової громадськості.

У проекті ENIAC брав участь Джои фон Нейман. Первинна версія ENIAC вимагала перемонтування дротів для зміни програми. У 1948 році машина була за рекомендацією фон Неймана забезпечена спеціальним пристроєм для зберігання програми, а один з регістрів був пристосований для виконання функцій лічильника команд. Переробка понизила швидкодію машини приблизно в шість разів, проте при цьому середня тривалість перепрограмування знизилася з декількох днів до декількох годин, так що зміна була визнана прогресивною.

Пізніше Джои фон Нейман покинув проект ENIAC, щоб очолити розробку комп'ютера IAS (Immediate Address Storage). Розробку комп'ютера почали у 1951 році, а повністю закінчили у 1952.

Деякі автори стверджують, що саме IAS став першою в історії машиною фон Неймана – термін, якому відповідають практично усі нині існуючі комп'ютери.

Під машиною фон Неймана розуміється обчислювальна машина, що має однорідний запам'ятовуючий пристрій, призначений як для зберігання даних, так і для зберігання команд програми. Архітектурний принцип фон Неймана іноді називають також принципом програми, що зберігається. Сам принцип був сформульований фон Нейманом в 1945 році в статті, присвяченій ще одному проекту, названому EDVAC.

Є відомості про те, що принцип програми, що зберігається, був сформульований раніше фон Неймана. Так, схожі принципи згадуються в патентній заявці Конрада Цузе датованій 1936 роком. Зустрічаються твердження і про те, що IAS був далеко не першою обчислювальною машиною, що зберігала програму в тому ж просторі пам'яті, що і дані. При цьому згадують такі проекти, як IBM SSEC (1948), Manchester SSEM (1948), BIN AC (1949) та інші.

Так або інакше, словосполучення «машина фон Неймана» є сталим терміном, використовуваним для позначення обчислювальних машин, що відповідають принципу програми, що зберігається.

Поява принципу програми, що зберігається, зробила можливою появу спочатку систем програмування, включаючою компілятори, а потім і операційних систем.

Появу архітектурного принципу фон Неймана слід вважати одним з найсерйозніших досягнень періоду лампових ЕОМ, що називаються традиційно ***ЕОМ першого покоління***.

За цей період про операційні системи ніхто і не чув. Програми безпосередньо взаємодіяли з апаратним забезпеченням машини. Ці комп'ютери управлялися безпосередньо з пульта управління (панелі), що складалась з сигнальних ламп і тумблерів. Звичайний режим роботи програміста був такий: записатися на певний час, потім спуститися до машинної кімнати (чи залу), вставити свою комутаційну панель в комп'ютер і проводити різні маніпуляції на панелі для розв'язання свого завдання за допомогою кнопок, перемикачів і індикаторів [5].

На початок 50-х років, з випуском перфокарт ситуація дещо покращилась. Стало можливим замість використання комутаційних панелей записувати і

прочитувати програми з перфокарт, але в усьому іншому процедура обчислень залишалася колишньою.

Такий режим роботи можна назвати ***послідовною обробкою даних***. Ця назва відображає той факт, що призначені для користувача програми виконувалися на комп'ютері послідовно. Через деякий час в спробі підвищити ефективність послідовної обробки були розроблені різні системні інструменти. До них належать бібліотеки функцій, редактори зв'язків, завантажувачі і драйвери введення-виведення, які існували у вигляді програмного забезпечення, яке було загальнодоступним для всіх користувачів.

1.4 Прості пакетні системи

Друге покоління (1955-1965): транзистори і системи пакетної обробки. В середині 50-х років винахід і застосування транзисторів радикально змінили всю картину. Комп'ютери стали досить надійними, з'явилася висока ймовірність того, що машина працюватиме досить довго, виконуючи при цьому корисні функції. Саме у цей період сталося розділення персоналу на програмістів і операторів, експлуатаційників і розробників обчислювальних машин. Обчислювальні машини, що побудовані на основі транзисторів, прийнято називати ***ЕОМ другого покоління***.

До цього періоду належать початок масового виробництва обчислювальних машин. Так, компанія Digital Equipment Corporation продала близько 50000 екземплярів комп'ютера PDP-8. До речі, ця машина знаменита ще і тим, що саме в її архітектурі був уперше застосований принцип ***загальної шини*** (архітектура фон Неймана).

До епохи комп'ютерів другого покоління належать такі важливі інновації, як поява ***мов програмування високого рівня*** (першою мовою програмування вважається Фортран, розроблений Бекусом, 1954-1957 рр.), і, нарешті, ***поява операційних систем***.

Машини, які називали ***мейнфреймами*** (від англ. *mainframe* – комп'ютер загального призначення зі значними ресурсами введення-виведення, великим об'ємом оперативної і зовнішньої пам'яті), розташовувалися в спеціальних кімнатах з кондиціонованим повітрям, де ними управляв цілий штат професійних операторів. Тільки великі корпорації могли дозволити собі техніку, ціна якої обчислювалася мільйонами доларів. Щоб виконати завдання, програміст спочатку повинен був записати його на спеціальні бланки (Фортрану або Асемблера), а потім перенести на перфокарти. Після цього принести колоду перфокарт до кімнати введення даних, передати операторові та йти чекати результатів. Потім оператор брав одну з колод перфокарт і вводив в комп'ютер.

Якщо враховувати високу вартість устаткування, не дивно, що люди досить скоро зайнялися пошуком способу підвищення ефективності використання машинного часу. Простої, що відбувалися із-за неузгодженості розкладу, а також час, витрачений на підготовку і розв'язання задачі, – усе це було занадто дорого. Загальноприйнятим рішенням стала ***система пакетної обробки***. Задум полягав у тому, щоб зібрати декілька завдань (колод перфокарт),

переписати їх на магнітну стрічку (барабан). Потім оператор завантажував спеціальну програму (прообраз сьогоднішньої ОС), яка прочитувала перше завдання із стрічки і запускала його. Вихідні дані видавалися або впродовж роботи, або також записувалися на стрічку, а потім окремо роздруковувалися на менш дорогому комп'ютері або спеціальному пристрої.

У ході реалізації систем пакетної обробки була розроблена формалізована ***мова управління завданнями,*** за допомогою якої програміст повідомляв систему, яку роботу він хоче виконати на обчислювальній машині. Сукупність декількох завдань, як правило, у вигляді колоди перфокарт, дістала назву ***пакету завдань***.

Головна ідея, що лежить в основі простих пакетних схем обробки, полягала у використанні спеціальної програми, відомої під назвою ***монітор*** (на вітчизняних машинах – ***диспетчер***). Використовуючи ОС такого типу, користувач не мав безпосереднього доступу до машини.

Робота з точки зору монітора виглядала таким чином. Монітор управляв послідовністю подій. Щоб це було можливо, основна його частина розташовувалася в основній пам'яті і завжди була готова до роботи. Цю частину називали ***резидентним монітором***. Іншу частину складали утиліти і загальні функції, які завантажувалися у вигляді підпрограм програмами користувача і монітором. Монітор прочитував з пристроїв введення-виведення поточне завдання і розміщував його в області пам'яті, призначеної для програм користувача, і передавав управління програмі. Після закінчення завдання воно повертало управління монітору, який відразу ж починав прочитувати наступне завдання.

З точки зору процесора він в деякий момент виконував команди монітора. Монітор після причитування завдання віддавав процесору команду переходу, за якою процесор повинен почати виконувати програму користувача. Процесор переходив до обробки програми користувача і виконував її команди до тих пір, поки не доходив до кінця або доки не виникала збійна ситуація.

Таким чином, монітор, або ***пакетна ОС***, – це звичайна комп'ютерна програма. Її робота була заснована на здатності процесора вибирати команди з різних областей пам'яті, при цьому відбувалася передача і повернення управління.

1.5 Багатозадачні пакетні системи

Але як би швидко і надійно не працювали оператори, вони ніяк не могли змагатися в продуктивності з роботою пристроїв комп'ютера. Велику частину часу процесор простоював в очікуванні, поки оператор запустить чергове завдання. А оскільки процесор був дуже дорогим пристроєм, то низька ефективність його використання означала низьку ефективність використання комп'ютера в цілому. Для розв'язання цієї проблеми були розроблені перші системи багатозадачної пакетної обробки, які автоматизували усю послідовність дій оператора з організації обчислювального процесу. Ранні системи пакетної обробки стали прообразом сучасних операційних систем. Вони стали першими

системними програмами, призначеними не для обробки даних, а для управління обчислювальним процесом. Але щоб реалізувати режим багатозадачної пакетної обробки, необхідно було при використанні монітора використати також і інші можливості апаратного забезпечення, такі як:

1. **Захист пам'яті**. Під час роботи програма користувача не повинна вносити зміни в область пам'яті монітора. Якщо це станеться, то апаратне забезпечення процесора повинне виявити помилку і передати управління монітору.
2. **Таймер**. Таймер використовувався для того, щоб запобігти ситуації, коли одне завдання захопить безмежний контроль над системою. Таймер виставлявся на початку кожного завдання, і після закінчення певного проміжку часу програма користувача зупинялася і управління передавалося монітору.
3. **Привілейовані команди**. Деякі команди повинні мати підвищені привілеї і виконуватися тільки монітором. Це, наприклад, команди введення-виведення. Якщо програмі користувача треба зробити уведення-виведення, вона повинна запросити для виконання цих операцій монітор.
4. **Переривання**. Ця можливість надає ОС велику гнучкість при передачі управління програмі користувача і його відновленні.

Постачальники комп'ютерів скоро зрозуміли, що без таких можливостей апаратного забезпечення навіть прості пакетні ОС можуть призвести до хаосу.

Третє покоління (1965-1980): інтегральні схеми і багатозадачність. В цей час у технічній базі обчислювальних машин стався перехід від окремих напівпровідникових елементів типу транзисторів до інтегральних мікросхем, що відкрило шлях до появи наступного покоління комп'ютерів. Великі функціональні можливості інтегральних схем зробили можливою реалізацію на практиці складної комп'ютерної архітектури.

У цей період були реалізовані практично усі основні механізми, властиві сучасним ОС: ***мультипрограмування, мультипроцесування, підтримка багатотермінального розрахованого на багато користувачів режиму, віртуальна пам'ять, файлові системи, розмежування доступу*** і ***мережева робота***.

У ці роки починається розквіт системного програмування. З напряму прикладної математики, що представляє інтерес для вузького кругу фахівців, системне програмування перетворюється на галузь індустрії, що робить безпосередній вплив на практичну діяльність мільйонів людей. Революційною подією цього етапу стала промислова реалізація мультипрограмування в двох варіантах – в системах пакетної обробки і розподілу часу.

На початок 60-х років, в основному, у світі переважали комп'ютери типу IBM-7094 для числових обчислень в науці і техніці, і IBM-1401, що широко використовувались банками для сортування і друкування даних. Розвиток і підтримка цих двох ліній для виробників були досить дорогим задоволенням. Тому фірма IBM спробувала вирішити ці проблеми разом, випустивши єдину серію універсальних машин IBM/360, що задовольняли потребам усіх покупців. Машини мали однакову структуру і набір команд, відрізнялися вони тільки

ціною і продуктивністю (об'ємом пам'яті, швидкодією процесора, кількістю пристроїв введення-виведення і тому подібне), і були програмно-сумісними.

Корпорація IBM добилася миттєвого успіху, а ідею сімейства сумісних комп'ютерів прийняли й усі інші основні виробники. Програмна сумісність вимагала і сумісності операційних систем. Такі операційні системи повинні були б працювати і на великих, і на малих обчислювальних системах, з великою і з малою кількістю різноманітної периферії, в комерційній області і в області наукових досліджень. Тому, щоб управляти усім цим сімейством комп'ютерів, була створена єдина операційна система OS/360, яка повинна була добре працювати на усіх цих моделях.

Ні IBM, ні інши фірми не могли написати програмне забезпечення (ПЗ), яке б задовольняло багатьох суперечливих вимог. У результаті з'явилася величезна і надзвичайно складна ОС. Вона складалася з мільйонів рядків, написаних на асемблері тисячами програмістів, і містила тисячі помилок.

Незважаючи на свої величезні розміри і недоліки, OS/360 і подібні до неї ОС 3-го покоління насправді непогано задовольняли вимоги клієнтів.

Іншим важливим плюсом ОС 3-го покоління стала здатність зчитувати завдання з перфокарт (або з стрічки, або диску) і записувати на диск у міру того, як їх приносили в машинний зал. Потім, як тільки закінчувалося поточне завдання, з диска (стрічки) прочитувалося наступне завдання. Цей технічний прийом називається ***«підкачуванням»*** даних або ***спулінгом*** (spooling, від Simultaneous Peripheral Operation On Line – спільна периферійна операція в інтерактивному режимі). З появою підкачування стали більше не потрібні спеціальні пристрої і багатократні перенесення магнітних стрічок.

Ефективність роботи процесора в ОС 3-го покоління значно підвищилася. Тепер в пам'яті комп'ютера було достатньо місця для операційної системи і двох або більше програм користувачів. Тепер, коли одне із завдань чекає завершення операцій введення-виведення, процесор може перемикатися на інше завдання і так далі. Такий режим відомий як ***багатозадачність*** і є основною рисою сучасних ОС.

Робота багатозадачної пакетної системи і деяких простих пакетних систем базувалася на деяких апаратних особливостях комп'ютера. Найбільш найзначнішим доповненням, корисним для багатозадачності, стало апаратне забезпечення, що підтримує переривання введення-виведення. Використовуючи цю можливість, процесор генерує команду введення-виведення для одного завдання і переходить до іншого на той час, поки контролером пристрою виконується введення-виведення. Після завершення операції введення-виведення процесор отримує переривання, і управління передається програмі обробки переривань із складу ОС. Потім управління передається іншому завданню. Багатозадачні ОС складніші за системи послідовної обробки завдань, хоч би через те, щоб обробити декілька завдань одночасно, вони повинні знаходитися в основній пам'яті, а для цього потрібна система ***управління пам'яттю***.

1.6 Системи з режимом розподілу часу

При системах 3-го покоління ще залишався різновид пакетної обробки. Багато програмістів сумували за першим поколінням машин, коли вони могли годинами розпоряджатися усією машиною і досить швидко відлагоджувати свої програми.

Бажання скоротити час очікування відповіді, і для того щоб хоч би частково повернути користувачам відчуття безпосередньої взаємодії з комп'ютером, був розроблений інший варіант мультипрограмних систем – системи ***розподілу часу***, варіант багатозадачності, при якому в кожного користувача є свій термінал.

Варіант мультипрограмування, який застосовувався в системах розподілу часу, був націлений на створення для кожного окремого користувача ілюзії одноосібного володіння обчислювальною машиною за рахунок періодичного виділення кожній програмі своєї частки процесорного часу. Якщо двадцять користувачів зареєстровані в системі, що працює в режимі розподілу часу, і сімнадцять з них думають, розмовляють або п'ють каву, то центральний процесор (ЦП) по черзі надається трьом користувачам, що працюють в цей час на машині, і забезпечує швидке інтерактивне обслуговування цих користувачів. При цьому він може працювати над великими пакетами у фоновому режимі, коли ЦП не зайнятий іншими завданнями.

Перша серйозна система з режимом розподілу часу **CTSS** (Compatible Time Sharing System – сумісна система розподілу часу) для управління обчислювальними ресурсами комп'ютера IBM 7090 була розроблена на початку 60-х років групою програмістів під керівництвом професора **Фернандо Корбато** в Массачусетському технологічному інституті (MTI). Головним розробником був математик **Джон Маккарти**.

Після успіху системи CTSS та ж група з MTI, дослідницька лабораторія Bell Labs і корпорація General Electric у 1965 почали розробку машини, яка повинна була підтримувати одночасно сотні користувачів в режимі розподілу часу. Проектувальники цієї системи, відомої як **MULTICS** (MULTiplexed Information and Computing Service – мультиплексна інформаційна і обчислювальна служба), уявляли собі одну величезну машину, скористатися послугами якої могла кожна людина в районі Бостона. Думка про те, що набагато потужніші машини, чим їх мейнфрейм GE-645, продаватимуться мільйонами за ціною тисяча доларів і менше за штуку усього лише через тридцять років, здавалася чистісінькою науковою фантастикою.

Існувало багато причин, в яких система MULTICS не захопила весь світ. Не останню роль зіграв той факт, що система була написана мовою високого рівня PL/1, а нормальний компілятор мови PL/1 з'явився лише через декілька років. У результаті, система MULTICS подала багато конструктивних ідей комп'ютерним теоретикам, але перетворити її на серйозний продукт і добитися комерційного успіху виявилося набагато важче, ніж очікувалося. Саме розробники MULTICS почали застосовувати термін «***процес***» в його сучасному значенні в контексті ОС.

Уперше в MULTICS почали застосовувати ***віртуальну пам'ять*** – метод управління пам'яттю комп'ютера, що дозволяє виконувати програми, що вимагають більше оперативної пам'яті, чим є в комп'ютері, шляхом автоматичного переміщення частин програми між основною пам'яттю і вторинним сховищем (наприклад, жорстким диском).

Але усі спроби налагодити в системі відносно дружній інтерфейс провалилися. Було вкладено багато грошей, а результат був дещо іншим, ніж хотілося б керівництву з Bell Labs. Проект був закритий. До речі, учасниками проекту значилися **Кен Томпсон** і **Денніс Рітчі**. Попри те, що проект був закритий, вважається, що саме ОС MULTICS дала початок ОС Unix.

До основних досягнень цього періоду в області розробки систем, що працюють в режимі розподілу часу, належать також TSS (Система з розподілом часу), розроблена компанією IBM, і CP/CMS (діалогова моніторна система, IBM), яка потім перетворилася на ОС віртуальних машин (VM). MULTICS, TSS і CP/CMS – усі ці системи мали вже віртуальну пам'ять.

За свій вклад в розробку CTSS і MULTICS в 1990 році Ф. Корбато був удостоєний престижної премії Т'юринга.

Незабаром Bell Labs вибула з проекту, а корпорація General Electric зовсім залишила комп'ютерний бізнес. Проте Массачусетський технологічний інститут проявив завзятість і з часом отримав працюючу систему. Вона була продана як комерційний виріб компанії Honeywell, що купила комп'ютерний бізнес в General Electric, і встановлена приблизно у вісімдесяти великих компаніях і університетах по всьому світу. Незважаючи на свою нечисленність, користувачі системи були відчайдушно віддані їй. Так, наприклад, компанії General Motors, Ford залишили свої системи в кінці 90-х років, через 30 років після їх виходу. Останній комп'ютер під управлінням MULTICS працював до 2000 року.

Ще одним важливим моментом розвитку за часів 3-го покоління було феноменальне зростання міні-комп'ютерів, починаючи з випуску PDP-1 корпорацією DEC у 1961 році. Ці комп'ютери мали оперативну пам'ять, що складалась усього лише з 4К 18-бітових слів, але коштували вони всього по 120 тисяч доларів (близько 5% від ціни IBM-7094). За цією машиною послідувала ціла серія інших, і як кульмінація – PDP-11.

У 1969 році **Кен Томпсон** (Ken Thompson), один з розробників MULTICS з Bell Labs, знайшов міні-комп'ютер PDP-7, яким ніхто не користувався, і вирішив написати усічену, розраховану на одного користувача, версію системи MULTICS. Пізніше ця система розвинулася в ОС UNIX, що стала популярною в академічному світі і в урядових установах. Системний час реалізації UNIX відлічують з 1 січня 1970 року.

Надалі ще один дослідник лабораторії Bell Labs, **Брайан Керніган**, якось жартома назвав цю систему **UNICS** (UNiplexed Information and Computing Service – примітивна інформаційна і обчислювальна служба). Призвисько, дане Керніганом, міцно пристало до нової системи, хоча написання цього слова стало злегка коротше при цій же вимові, перетворившись на **UNIX**.

Робота Томпсона справила на його колег з Bell Labs таке сильне враження, що незабаром до нього приєднався **Денніс Рітчі**, а трохи пізніше і увесь його

відділ. Незабаром система UNIX була перенесена на потужніші комп'ютери PDP-11/20,45,70.

У 1974 році Рітчі і Томпсон опублікували статтю про ОС UNIX, яка стала важливою віхою в історії комп'ютерів. За роботу, описану в цій статті, їм пізніше асоціацією з обчислювальної техніки ACM була присуджена премія Т'юрінга.

Експериментальні системи з розподілом часу в 70-х роках перетворилися на успішний комерційний продукт. Зв'язок між комп'ютерними системами США ставав все інтенсивнішим у міру того, як усе ширше застосування отримували комунікаційні стандарти TCP/IP Міністерства Оборони. Зв'язок локальною мережею став можливим і недорогим завдяки стандарту Ethernet, розробленому компанією Xerox.

Зросла і кількість проблем, пов'язаних з безпекою, завдяки збільшенню об'ємів інформації, що передається уразливими лініями зв'язку. Величезну увагу почали приділяти шифруванню, украй необхідним стало кодування персональної і захищеної правами власності інформації.

Багатотермінальний режим використовувався не лише в системах розподілу часу, але і в системах пакетної обробки. При цьому не лише оператор, але і усі користувачі одержували можливість формувати свої завдання і управляти їх виконанням зі свого терміналу. Такі операційні системи одержали назву ***систем віддаленого введення завдань.***

Термінальні комплекси могли розташовуватися на великій відстані від процесорних стійок, з'єднуючись з ними за допомогою різних глобальних зв'язків – модемних з'єднань телефонних мереж або виділених каналів. Для підтримки віддаленої роботи терміналів в операційних системах з'явилися спеціальні програмні модулі, що реалізовують різні протоколи зв'язку. Такі обчислювальні системи з віддаленими терміналами, зберігаючи централізований характер обробки даних, якоюсь мірою були прообразом сучасних мереж, а відповідне системне програмне забезпечення – прообразом ***мережевих операційних систем***.

Декілька слів хотілося б сказати про **системи реального часу**, що застосовуються для управління різними технічними об'єктами, такими, наприклад, як верстат, супутник і тому подібними. В усіх цих випадках існує гранично допустимий час, впродовж якого має бути виконана та або інша програма, що управляє об'єктом, інакше може статися аварія (наприклад, супутник вийде із зони видимості).

Таким чином, критерієм ефективності для систем реального часу є їх здатність витримувати заздалегідь задані інтервали часу між запуском програми і отриманням результату (керуючої дії). Цей час називається ***часом реакції системи***, а відповідна властивість системи – ***реактивністю***. Для цих систем мультипрограмна суміш є фіксованим набором заздалегідь розроблених програм, а вибір програми на виконання здійснюється виходячи з поточного стану об'єкту або відповідно до розкладу планових робіт.

1.7 Операційні системи і глобальні мережі

На початку 70-х років з'явилися перші мережеві операційні системи, які на відміну від багатотермінальних ОС дозволяли не лише розосередити користувачів, але і організувати розподілене зберігання і обробку даних між декількома комп'ютерами, пов'язаними електричними зв'язками. Будь-яка мережева операційна система, з одного боку, виконує усі функції локальної операційної системи, а з іншого боку, має деякі додаткові функції, що дозволяють їй взаємодіяти по мережі з операційними системами інших комп'ютерів.

Програмні модулі, що реалізовують мережеві функції, з'являлися в операційних системах поступово, у міру розвитку мережевих технологій, апаратної бази комп'ютерів і виникнення нових задач, що вимагають мережевої обробки.

Хоча теоретичні роботи зі створення концепцій мережевої взаємодії велися майже з самої появи обчислювальних машин, значимі практичні результати з об'єднання комп'ютерів в мережі були отримані в кінці 60-х. За допомогою глобальних зв'язків і техніки комутації пакетів вдалося реалізувати взаємодію машин класу мейнфреймів і суперкомп'ютерів. Ці дорогі комп'ютери часто зберігали унікальні дані і програми, доступ до яких необхідно було забезпечити широкому колу користувачів, що знаходилися в різних містах на значній відстані від обчислювальних центрів.

Ще в лютому 1958 року в США було створено потужне відомство, що зіграло ключову роль в історії комп'ютерних мереж. Перед цим відомством було поставлено завдання забезпечити стратегічну перевагу США у сфері високих технологій. Воно працювало при міністерстві оборони і називалося Управлінням перспективних досліджень (Advanced Research Projects Agency – **ARPA**).

У 1969 році Міністерство оборони США ініціювало роботи по об'єднанню суперкомп'ютерів оборонних і науково-дослідних центрів в єдину мережу. Ця мережа дістала назву ARPAnet і стала відправною точкою для створення найвідомішої нині глобальної мережі – Інтернету. Мережа ARPAnet об'єднувала комп'ютери різних типів, що працювали під управлінням різних ОС з доданими модулями, які реалізовують комунікаційні протоколи, загальні для усіх комп'ютерів мережі.

У 1974 році компанія IBM оголосила про створення власної мережевої архітектури для своїх мейнфреймів, що отримала назву SNA (System Network Architecture). Ця багаторівнева архітектура, багато в чому подібна до стандартної моделі OSI, що з'явилася дещо пізніше, забезпечувала взаємодію типу «термінал-термінал», «термінал-комп'ютер» і «комп'ютер-комп'ютер» глобальними зв'язками. Нижні рівні архітектури були реалізовані спеціалізованими апаратними засобами, найважливішим з яких є процесор телеобробки. Функції верхніх рівнів SNA виконувалися програмними модулями. Один з них складав основу програмного забезпечення процесора телеобробки. Інші модулі працювали на центральному процесорі в складі стандартної операційної системи IBM для мейнфреймів.

В цей же час в Європі велися активні роботи із створення і стандартизації мереж X.25 (***Рекомендація X.25***). Ці мережі з комутацією пакетів не були прив'язані до якої-небудь конкретної операційної системи. Після отримання статусу міжнародного стандарту в 1974 році протоколи X.25 стали підтримуватися багатьма операційними системами. З 1980 року компанія IBM включила підтримку протоколів X.25 в архітектуру SNA і у свої операційні системи.

1.8 Операційні системи міні-комп'ютерів

До середини 70-х років разом з мейнфреймами широке поширення отримали міні-комп'ютери, такі як PDP-11, HP (Hewlett Packard). Міні-комп'ютери першими використали переваги великих інтегральних схем, що дозволили реалізувати досить потужні функції при порівняно невисокій вартості комп'ютера.

Архітектура міні-комп'ютерів була значно спрощена в порівнянні з мейнфреймами, що знайшло відображення і в їх операційних системах. Багато функцій мультипрограмних ОС мейнфреймів, розрахованих на багато користувачів, було усічено, враховуючи обмеженість ресурсів міні-комп'ютерів.

Операційні системи міні-комп'ютерів часто стали робити спеціалізованими. Наприклад, операційні системи тільки для управління в реальному часі або тільки для підтримки режиму розподілу часу (RSX-11M для тих же комп'ютерів). Ці операційні системи не завжди були розрахованими на багато користувачів, що в багатьох випадках виправдовувалося невисокою вартістю комп'ютерів.

Важливою віхою в історії міні-комп'ютерів і в історії операційних систем явилося створення ОС UNIX. Спочатку ця ОС призначалася для підтримки режиму розподілу часу в міні-комп'ютері PDP-7. З середини 70-х років почалося масове використання ОС UNIX. До цього часу програмний код для UNIX був на 90% написаний мовою високого рівня C.

Широке поширення ефективних C-компіляторів зробило UNIX унікальною для того часу ОС, яка мала можливість порівняно легкого перенесення на різні типи комп'ютерів. Оскільки ця ОС поставлялася разом з початковими кодами, то вона стала першою відкритою ОС, яку могли удосконалювати прості користувачі-ентузіасти. Хоча UNIX була спочатку розроблена для міні-комп'ютерів, гнучкість, елегантність, потужні функціональні можливості і відкритість дозволили їй зайняти міцні позиції в усіх класах комп'ютерів: суперкомп'ютерах, мейнфреймах, міні-комп'ютерах, серверах і робочих станціях на базі RISC-процесорів, персональних комп'ютерах.

Доступність міні-комп'ютерів і як результат цього їх поширеність на підприємствах послужили потужним стимулом для створення локальних мереж. Підприємство могло собі дозволити мати декілька міні-комп'ютерів, що знаходяться в одній будівлі або навіть в одній кімнаті. Природно, виникала потреба в обміні інформацією між ними і в спільному використанні дорогого периферійного устаткування.

Перші локальні мережі будувалися за допомогою нестандартного комунікаційного устаткування, в простому випадку – шляхом прямого з'єднання послідовних портів комп'ютерів. Програмне забезпечення також було нестандартним і реалізовувалося у вигляді призначених для користувача застосувань. Перше мережеве застосування для ОС UNIX – програма UUCP (UNIX-to-UNIX Copy program) – з'явилася в 1976 році і почала поширюватися з версією 7 AT&T UNIX з 1978 року. Ця програма дозволяла копіювати файли з одного комп'ютера на інший в межах локальної мережі через різні апаратні інтерфейси – RS-232 і, крім того, могла працювати через глобальні зв'язки, наприклад модемні.

1.9 Основні досягнення

Четверте покоління комп'ютерів (з 1980 року): персональні комп'ютери. Наступний період в еволюції ОС пов'язаний з появою Великих Інтегральних Схем (ВІС або LSI – Large Scale Integration) – кремнієвих мікросхем. Початок 80-х років пов'язаний з ще одною знаменною для історії операційних систем подією – появою ***персональних комп'ютерів*** [5].

З точки зору архітектури персональні комп'ютери (що спочатку називалися ***мікрокомп'ютерами***) були багато в чому схожі на міні-комп'ютери класу PDP-11, але відрізнялися за ціною, що дозволило відділам компаній і факультетам університетів мати власні комп'ютери. Комп'ютери стали широко використовуватися неспеціалістами, що зажадало розробки «дружнього» програмного забезпечення, і надання цих «дружніх» функцій стало прямим обов'язком операційних систем. Персональні комп'ютери послужили також потужним каталізатором для бурхливого зростання локальних мереж, створивши для цього відмінну матеріальну основу у вигляді десятків і сотень комп'ютерів, що належали одному підприємству і розташованих в межах однієї будівлі.

У 1974 році компанія Intel випустила Intel 8080 – перший універсальний 8-розрядний процесор. Для цього процесора знадобилася ОС, за допомогою якої можна було б протестувати новинку. Компанія Intel залучила до розробок і написання потрібної ОС одного зі своїх консультантів **Гарі Кілдолла** (Gary Kildall). Спочатку Кілдолл з другом сконструювали контролер для 8-дюймового гнучкого диска, нещодавно випущеного компанією Shugart Associates, і підключили цей диск до процесора Intel 8080. Таким чином, з'явився перший мікрокомп'ютер з диском. Потім Кілдолл створив дискову операційну систему, названу **CP/M** (Control Program for Microcomputers).

У 1977 році компанія Digital Research переробила CP/M, щоб зробити цю систему придатною для роботи на мікрокомп'ютерах не лише з процесорами Intel 8080, а також з іншими процесорами. Потім було написано багато прикладих програм, які працювали в CP/M, що дозволило CP/M зайняти високу позицію у світі мікрокомп'ютерів упродовж 5 років.

На початку 80-х корпорація IBM розробила IBM PC і почала шукати для нього програмне забезпечення. Фірма IBM почала вісті переговори з Кілдоллом про розробку нової ОС. Але після того, коли він полетів у двотижневу відпустку,

співробітники IBM зв'язалися з Білом Гейтсом (Bill Gates), щоб отримати ліцензію на право використання інтерпретатора мови Бейсик. Вони також поцікавилися, чи не знає він ОС, яка працювала б на PC. Гейтс порадив звернутися до Digital Research. Фірма IBM почала вести переговори з Кілдоллом про розробку нової ОС, але Кілдолл запропонував перенести переговори на час після його відпустки (за іншими відомостями – він відмовився підписати строгі договори про нерозголошування, як того вимагала IBM). Корпорація IBM знову звернулася до Гейтса з проханням забезпечити її операційною системою.

Після повторного запиту IBM Гейтс з'ясував, що у місцевого виготівника комп'ютерів Seattle Computer Products (SCP) є відповідна ОС 86-**DOS.** У 1980 році компанії SCP знадобилася ОС для їх модуля пам'яті, орієнтованого на новий 16-розрядний процесор Intel 8086. Вона попросила **Тіма Патерсона** (Tim Paterson) написати нову ОС, яку назвали **QDOS** (Quick and Dirty Operating System – у буквальному перекладі це звучить, як «швидка і брудна ОС»). У повній відповідності з назвою, вона була написана всього за декілька місяців, але не була повністю протестована і відлагоджена. До кінця року Патерсон удосконалив свою ОС, яка була випущена як 86-DOS.

При створенні ОС 86-DOS переслідувалися декілька цілей. Вона мала бути сумісною з програмами, написаними для ОС CP/M, щоб дати можливість використати безліч прикладних програм. Тому 86-DOS повинна була мати той же API (Application Programming Interface – інтерфейс прикладного програмування). Щоб зробити 86-DOS ефективнішою, ніж CP/M, Патерсон написав її на асемблері і вбудував в неї модуль роботи з накопичувачами FAT.

Біл Гейтс запропонував компанії SCP викупити DOS за 50 тис. доларів. Коли корпорація IBM зажадала деяких вдосконалень в програмі, Біл Гейтс запросив для цієї роботи Тіма Патерсона. Фірма IBM уклала контракт з компанією MicroSoft, яку очолював Біл Гейтс, на розробку ОС.

Нова видозмінена система, перейменована в **MS-DOS** (MicroSoft Disk Operation System), була добре прийнята багатьма виробниками персональних комп'ютерів, сумісних з IBM PC. Система MS-DOS, створена компанією Microsoft, узурпувала місце, яке раніше належало CP/M, завоювавши до того ж ширший ринок.

Коли в 1983 році з'явився IBM PC/AT з центральним процесором Intel 80286, система MS-DOS вже міцно стояла на ногах. Пізніше система MS-DOS широко використовувалася на комп'ютерах з процесорами 80386 і 80486. Хоча первинна версія MS-DOS була досить примітивна, подальші версії системи виходили зі все краще розробленими властивостями, включаючи багато чого, запозиченого від CP/M і UNIX.

CP/M, MS-DOS і інші ОС для перших мікрокомп'ютерів повністю грунтувалися на введенні команд з клавіатури. Потім, завдяки дослідженням, проведеним в 60-і роки **Дагом Енгельбартом** (Doug Engelbart), ця властивість ОС змінилася. Енгельбарт винайшов ***графічний інтерфейс користувача*** (GUI, Graphical User Interface), що складався з вікон, значків, різних меню і миші. Цю ідею перейняли розробники з Xerox і вбудували в сконструйовані ними машини.

Коли Стів Джобс (Steve Jobs), той самий, який винайшов комп'ютер Apple, відвідав компанію Xerox, де побачив GUI і усвідомив його потенційну цінність, відразу ж приступив до створення Apple з графічним інтерфейсом. Це привело до проекту Lisa, який був занадто дорогий і потерпів комерційну невдачу.

Друга спроба Джобса, **Apple Macintosh**, мала величезний успіх, не лише через дешевизну, але і тому, що на ньому працював ***дружній інтерфейс***, тобто призначений для користувачів, які нічого не знають про комп'ютери і, більше того, зовсім не бажають чому-небудь навчатися. На 1989 рік операційна система Macintosh MacOS не мала собі рівних з простоти використання. Правда, до цього часу фірмі IBM вдалося власними зусиллями колективом з 5000 програмістів за 5 років розробити ОС OS/2, що містила близько 1 млн рядків коду (згодом OS/2 Warp, як і Windows NT, буде сучасною багатозадачною багатопотоковою ОС).

Коли корпорація Microsoft вирішила створити нову ОС, яка б замінила MS-DOS, вона перебувала повністю під впливом успіхів компанії Macintosh. Історія нової ОС бере свій початок у 1985 році, коли з'явилася перша версія системи під назвою Windows 1.0, базою для якої послужив GUI. Система Windows 1.0 спочатку працювала поверх MS-DOS. Наприклад, в Windows 1.0 вікна не могли перекриватися, що було вже усунено в Windows 2.0.

Найважливишою особливістю Windows 2.0 стала поява підтримки захищеного режиму (protected mode) для програм DOS, який забороняв програмам записувати дані в простір пам'яті інших програм, завдяки чому була підвищена стабільність роботи системи. Упродовж 10 років, з 1985 по 1995 рік, система Windows виконувала роль графічного середовища поверх MS-DOS. За цей час при розробці нових версій усі зусилля були спрямовані на підвищення надійності, а також на підтримку засобів мультимедіа (версія 3.1) і роботу в комп'ютерних мережах (версія 3.11). Так, в Windows 3.1 з'явився ***розширений режим*** (enhanced mode), який дозволив додаткам Windows використати більше пам'яті, ніж могли собі дозволити програми DOS.

Проте в 1995 році вийшла автономна версія Windows 95, що стала новим етапом в історії Windows. Windows 95 включала багато особливостей ОС MS-DOS, але тільки для завантаження і виконання старих програм.

В порівнянні з Windows 3.1 значно змінився інтерфейс, збільшилась швидкість роботи програм. Однією з нових можливостей Windows 95 була можливість автоматичного налаштування додаткового устаткування комп'ютера для роботи без конфліктів один з одним. Іншою важливою особливістю системи стала можливість роботи з Інтернетом без використання додаткових програм.

Продовженням розвитку Windows 95 стала злегка змінена версія цієї системи – Windows 98, що з'явилася в 1998 році. Проте і Windows 95, і Windows 98 все ще містили велику кількість програм 16-розрядного асемблера Intel.

Паралельно з розробкою Windows компанія Microsoft у 1988 році почала роботу над новою ОС, названою **Windows NT** (NT означає New Technology – нова технологія), яка на певному рівні сумісна з Windows 95, але її ядро було написане повністю наново. Під впливом ОС OS/2 корпорацією Microsoft спочатку була зроблена невдала спроба розробити цю систему спільно з фірмою IBM. Але потім, ця повністю 32-розрядна система, була розроблена самостійно.

Девід Кетлер (David Catler), головний розробник Windows NT, був також одним з творців ОС VMS для комп'ютерів PDP-11 і VAX. У 1988 році він прийняв від Біла Гейтса пропозицію стати одним з керівників проекту розробки наступника OS/2 – Windows NT. Тому деякі ідеї VMS є присутніми і в Windows NT. Кетлер настояв на прийнятті 20 розробників з корпорації DEC, де розроблялася ОС VAX, щоб сформувати нову команду.

Система Windows NT використовувала всі можливості сучасних мікропроцесорів і забезпечувала багатозадачність з одним користувачем або в багатокористувацькому середовищі. У ній уперше була реалізована підтримка інтерфейсу Win32, проте Кетлеру довелося частково займатися і підтримкою API DOS, POSIX і OS/2 на додаток до забезпечення виконання програм для Windows 3.0. Натхнення успіхом ОС Mach на основі мікроядра, команда Кетлера розробила нову ОС, що має досить компактне ядро з багаторівневою модульною структурою для обробки різних інтерфейсів. Корпорація Microsoft очікувала, що вже перша версія NT витіснить MS-DOS і усі інші версії Windows, оскільки це була система, що набагато перевершувала попередні, але надія не виправдалася. І тільки системі Windows NT 4.0 нарешті вдалося відносно широке поширення, особливо в корпоративних мережах.

Нова версія Windows NT 5.0 в 1999 році була перейменована в Windows 2000 (скорочений варіант – **W2K**, 2K – це, власне, і є 2000: «два кіло»). Єдина річ, якої немає в Windows 2000, – це MS-DOS. Її просто не було ні в якому вигляді (як і не було в NT). Був інтерфейс командного рядка, але це була нова 32-розрядна програма, що включає функціональність старої системи MS-DOS. 16-розрядного коду в NT дійсно немає, але це не заважає запускати в NT і Windows 2000 більшість старих 16-розрядних програм. Для цього в системі міститься спеціальна система емуляції 16-розрядної машини. Windows 2000 успадкувала від NT високу надійність і захищеність інформації від стороннього втручання.

Windows 2000 повинна була стати наступником і Windows 98, і Windows NT 4.0. Але цьому також не судилося статися, тому корпорація Microsoft випустила ще одну версію Windows 98, названу **Windows Me** (Millenium edition – випуск тисячоліття). В порівнянні з Windows 98 **Me** набула багато нових можливостей. Передусім, це удосконалена робота із засобами мультимедіа, можливість записувати не лише аудіо, але й відеоінформацію, потужні засоби відновлення інформації після збоїв і багато що інше.

Поступово різниця між різними системами Windows стирається, і на арену виходить нова операційна система версії NT 5.1, відоміша як **Windows XP**, призначена для заміни як Windows 2000, так і Windows Me.

Букви **XP** в назві нової версії популярної ОС Windows є частиною англійського слова **eXPerience**, яке перекладається як життєвий досвід, знання. При створенні операційної системи Windows XP був використаний багаторічний досвід розробників найпопулярніших комп'ютерних програм і систем, а також знання, накопичені в результаті спілкування з численними користувачами. Без сумніву, нова версія Windows була значним кроком вперед, в порівнянні з попередніми версіями.

1.10 Сучасні системи UNIX

Головним суперником Windows у світі персональних комп'ютерів все ж була і залишається система UNIX (і її різні похідні). UNIX є найсильнішою системою для робочих станцій і інших комп'ютерів старших моделей, таких як мережеві сервери. Вона стала особливо популярна на машинах з високопродуктивними RISC-процесорами (RISC, Reduced Instruction Set Computer – комп'ютер із скороченим набором команд).

В процесі розвитку ОС UNIX з'явилося багато її реалізацій, кожна з яких має свої корисні можливості. Згодом виникла необхідність створити реалізацію, в якій було б уніфіковано багато важливих нововведень, додані можливості інших сучасних ОС. Розглянемо деякі приклади сучасних ОС UNIX.

System V Release 4(SVR4). Версія SVR4 (1989 рік), розроблена спільно компаніями AT&T і Sun Microsystems, поєднує в собі особливості версій SVR3, Microsoft Xenix System і SunOS. Внаслідок чого з'явилася очищена від усього зайвого, хоча і складна в реалізації ОС. Серед нових можливостей цієї версії слід зазначити підтримку обробки даних в реальному часі, наявність класів планування процесів, динамічно розподілювані структури даних, управління віртуальною пам'яттю, наявність ядра з витісненням.

Solaris 4.x. Система Solaris (правильніше говорити SunOS 4.x) – це версія ОС UNIX, що розроблена фірмою Sun на основі SVR4. На час цього огляду останньою версією, що вийшла, була Solaris 4.x. Solaris версії 2.x мають усі можливості системи SVR4, а також деякі додаткові, такі як повне витеснення, наявність багатопотокового ядра, напівфункціональна підтримка SMP (Symmetric MultiProcessing – технологія ***симетричної багатопроцесорності***) і об'єктно-орієнтований інтерфейс файлових систем. Solaris – це найширше застосовувана реалізація ОС UNIX, що користується комерційним успіхом.

1.11 Маленький UNIX виростає в LINUX

У 1985 році був створений Portable Operating System Interface for Computing Environment, скорочено **POSIX** (переносний інтерфейс операційної системи для обчислювального середовища). На сьогодні більшість операційних систем задовольняють (повністю або частково) стандарту POSIX.

Linux – це сучасна POSIX-сумісна і Unix-подібна 32-х розрядна (64-х розрядна на платформі DEC AXP) операційна система для персональних комп'ютерів і робочих станцій. Це розрахована на багато користувачів мережева операційна система з мережевою віконною графічною системою **X Window System**. ОС Linux підтримує стандарти відкритих систем і протоколи мережі Internet, і сумісна з системами Unix, DOS, MS Windows. Усі компоненти системи, включаючи початкові тексти, поширюються з ліцензією на вільне копіювання і установку для необмеженого числа користувачів.

ОС Linux широко поширена на платформах Intel PC 386/486/Pentium/Pentium Pro і завойовує позиції на ряду інших платформ (DEC AXP, Power Macintosh та ін.).

1.11.1 Історія створення Linux

Спочатку розробка ОС Linux була виконана **Лінусом Торвалдсом** (Linus Torvalds), 21-річним студентом університету з Хельсінкі, як хобі. Його надихнула операційна система **Minix –** маленька UNIX-система, створена **Енді Таненбаумом** (Andy Tanenbaum). Він став серйозно обмірковувати маніакальну ідею, як зробити ОС Minix краще. Потім ОС Linux розвивалася великою командою з тисячі користувачів мережі Internet, співробітників дослідницьких центрів, фондів, університетів тощо.

5 жовтня 1991 року Лінус оголосив першу «офіційну» версію Linux, версія 0.02. Після версії 0.03 Лінус стрибком перейшов в нумерації до версії 0.10, оскільки над проектом стало працювати багато програмістів. Після декількох переглядів версій, Линус присвоїв черговій версії номер 0.95, щоб тим самим показати своє враження про те, що скоро можлива вже «офіційна» версія. (Зазвичай програмам не дають номер версії 1.0 до того, як вона теоретично завершена і відлагоджена). Це було в березні 1992 року. Приблизно через півтора року – в грудні 1993 версія ядра все ще була Linux 0.99.pl14 – асимптотично наближаючись до 1.0. Потім послідували версія ядра 1.1 patchlevel 52 і версія 1.2.

Лінус також вирішив, що ОС Linux повинна відповідати специфікаціям POSIX для забезпечення здатності до взаємодії з іншими UNIX-подібними системами. Згадаємо, що POSIX (Portable Operating System Intreface – інтерфейс переносних операційних систем) визначає стандарти прикладних інтерфейсів служб ОС.

Сьогодні Linux – це повноцінна ОС сімейства UNIX, здатна працювати з X Windows, TCP/IP, Emacs, UUCP, mail і USENET. Практично всі найважливіші програмні пакети поставлені і на Linux, тобто для Linux тепер доступні і комерційні пакети. Все більша різноманітність устаткування підтримується в порівнянні з первинним ядром.

1.11.2 Системні характеристики Linux

ОС Linux підтримує більшість властивостей, притаманних іншим реалізаціям UNIX, а також ряд таких, яких не було в інших ОС. Linux – це повна багатозадачна ОС, яка розрахована на багато користувачів.

ОС Linux досить добре сумісна з рядом стандартів для UNIX на рівні початкових текстів, включаючи IEEE POSIX.1, System V і BSD. Крім того, усі початкові тексти для Linux, включаючи ядро, драйвери пристроїв, бібліотеки, програми користувача і інструментальні засоби поширюються вільно.

Linux підтримує різні типи файлових систем для зберігання даних. Деякі файлові системи, такі як файлова система *ext2fs*, були створені спеціально для Linux. Підтримуються також інші типи файлових систем, такі як Minix-1 і Xenix. Реалізована також файлова система MS-DOS, що дозволяє прямо звертатися до файлів MS-DOS на жорсткому диску. Підтримується також файлова система ISO 9660 CD-ROM для роботи з дисками CD-ROM.

Linux забезпечує повний набір протоколів TCP/IP для мережевої роботи. Це включає драйвери пристроїв для багатьох популярних карт Ethernet, SLIP (Serial Line Internet Protocol, що забезпечують доступ по TCP/IP при послідовному з'єднанні), PLIP (Parallel Line Internet Protocol), PPP (Point - to - Point Protocol), NFS (Network File System) тощо.

Ядро Linux було відразу створене з урахуванням спеціального захищеного режиму для процесорів Intel 80386 і 80486. Ядро Linux підтримує завантаження тільки потрібних сторінок. Тобто з диска в пам'ять завантажуються ті сегменти програми, які дійсно використовуються. Можливе використання однієї сторінки, фізично один раз завантаженої в пам'ять, декількома виконуваними програмами.

Для збільшення об'єму доступної пам'яті Linux здійснює також розбиття диска на сторінки: тобто на диску може бути виділено до 256 Мб ***«простору для свопінгу»*** (swap space). Коли системі потрібна більше фізичної пам'яті, то вона за допомогою свопінгу виводить неактивні сторінки на диск. Це дозволяє виконувати об'ємніші програми і обслуговувати одночасно більше користувачів. Проте свопінг не виключає нарощування фізичної пам'яті, оскільки він знижує швидкодію, збільшує час доступу. Ядро підтримує універсальний пул пам'яті для програм користувача і дискового кешу. При цьому для кешу може використовуватися вся пам'ять, і навпаки, кеш зменшується при роботі великих програм.

Система **X Window** (чи просто – X) – стандартний графічний інтерфейс для UNIX-машин. Це потужне середовище, що підтримує багато додатків. Використовуючи X Window, користувач може одночасно мати на екрані декілька вікон, при цьому кожне має незалежний login. Часто використовується миша, хоча вона необов'язкова. Було написано багато специфічних X-додатків, таких як ігри, графічні утиліти, інструментарій для програмування тощо.

Ось деякі можливості, які має ОС Linux:

- ОС Linux дає можливість безкоштовно і легально мати сучасну ОС для використання як на роботі, так і вдома;
- має високу швидкодію;
- працює надійно, стійко, абсолютно без зависань;
- стійка до вірусів;
- ефективно управляє багатозадачністю і пріоритетами, фонові завдання (тривалі обчислення, передача електронної пошти модемом, форматування дискети тощо) не заважають інтерактивній роботі;
- дозволяє легко інтегрувати комп'ютер в локальні і глобальні мережі, у тому числі в Internet;
- працює з мережами на базі Novell і MS Windows;
- дозволяє виконувати представлені у форматі завантаження прикладні програми інших ОС – різних версій Unix, DOS і MS Windows;
- забезпечує використання величезного числа різноманітних програмних пакетів, накопичених у світі Unix і вільно поширюваних разом з початковими текстами.

1.11.3 Відмінності між Linux і іншими ОС

Свого часу MS-DOS не використала повністю функціональні можливості 80386 і 80486 процесорів. З іншого боку, Linux повністю працює в захищеному режимі процесора і реалізує усі можливості процесора. Linux забезпечує повний UNIX-інтерфейс, відсутній в MS-DOS. На Linux можна писати і відлагоджувати прикладні програми для UNIX.

Існує інструментарій, що дозволяє взаємодіяти Linux, MS-DOS і Windows 9.x. Наприклад, просто отримати доступ до файлів MS-DOS з Linux. Є також емулятори MS-DOS і Windows 9.x, що дозволяють виконувати багато популярних прикладних пакетів MS-DOS і Microsoft Windows.

Ряд інших ОС. таких як OS/2 фірми IBM і Windows NT фірми Microsoft, стають усе популярніші, у міру відходу користувачів з MS-DOS.

І OS/2, і Windows NT є повними багатозадачними операційними системами, як і Linux. Чисто технічно OS/2, Windows NT і Linux дуже схожі. Вони мають схожі інтерфейси користувача, систему захисту і тому подібне. Але головна відмінність полягає в тому, що Linux є різновидом UNIX, а звідси усі переваги належності до UNIX-співтовариства.

UNIX – це не лише найпопулярніша операційна система для розрахованих на багато користувачів машин, це також база для більшої частини програм, що вільно поширюються у світі.

1.11.4 Розвиток системи Linux

У університеті Карнеги-Меллона був розроблений проект **Mach –** мікроядерна архітектура операційної системи (1985-1994 рр). Проектом керував **Річард Ф. Рашид**, головний віце-президент компанії Microsoft з досліджень. ОС Mach була включена в пізніші системи, такі як Mac OS X, NeXT, а також значною мірою вплинула на Windows N і на Windows XP.

Мікроядро ОС Mach управляє процесами, обміном повідомлень між ними, віртуальною пам'яттю і драйверами пристроїв. Інша частина ОС реалізується у вигляді серверів – програм, які виконуються в призначеному для користувача режимі. Зокрема, це означає, що користувач може замінити сервер власною реалізацією.

Мікроядро – це сучасна технологія, що орієнтована на роботу на багатопроцесорних системах, має високу міру незалежності від апаратної платформи і пристосовність під потреби користувача.

З 3 по 6 листопада 1994 р. в Бухаресті відбулася друга Румунська конференція з Відкритих Систем (ROSE'94), на якій виступав Річард Століман (Richard Stollman), засновник і президент фонду Free Software Foundation (FSF – це організація програмістів). Він повідомив про поточний стан проекту Hurd.

Hurd – це вільна ОС, яка реалізована у вигляді серверів над мікроядром Mach як розширений варіант Unix. Hurd – це завершальна стадія проекту Hurd – створення вільного стандартного середовища ОС Unix, – який розробляє фонд FSF. Ось що говорить Луї-Домінік Дюбо (Louis-Dominique Dubeau) розробник

сервера файлової системи Linux: «Hurd відмінно спроектований і, думаю, виправдає очікування. Використання технології Hurd для того щоб реалізувати Linux на базі мікроядра – це краще з можливих рішень у наш час». Таким чином, Linux і Hurd йдуть один назустріч одному. Це сервери над мікроядром Mach.

Фірма Apple спонсорує розробку MkLinux (Linux on the OSF Microkernel) – сервера Linux над мікроядром OSF. Розробку виконує OSF Research Institute. Усі початкові тексти для платформ Intel і Power Macintosh поширюються вільно. Сервер Linux поширюється під ліцензією GNU GPL, мікроядро і інші сервери поширюються під ліцензією OSF Free Copyright. Нині версію MkLinux на платформі Intel і Power Macintosh можна отримати через мережу Internet.

У найзагальнішій постановці задачею FSF є усунення обмежень з копіювання, поширення, вивчення і модифікації програм для комп'ютерів. Для досягнення цього загального завдання FSF стимулює розробку і використання вільного програмного забезпечення, орієнтованого на широкий клас додатків. Конкретніше, FSF веде розробку програм у рамках проекту GNU (абревіатура GNU розкривається рекурсивно – GNU's Not Unix). Метою проекту GNU є створення повної інтегрованої програмної системи, засоби якої сумісні з можливостями середовища ОС UNIX.

У 2011 році ОС Linux відмічала свій ювілей. Щоб оцінити увесь 20-річний шлях ОС Linux, портал Business Insider публікує надані Linux Foundation дані, платформи, що наочно демонструють досягнення.

Згідно Linux Foundation, з 1992 по 2010 роки число розробників, що використовують ядро Linux, збільшилося **десятиразово**. Крім того, з моменту створення значно збільшились розміри програмного коду ядра Linux. Якщо в 1995 році в ньому налічувалося 250 тисяч рядків, то в 2010 році їх число збільшилось до 14 мільйонів.

Платформа значно посилила свої позиції в різних областях, зокрема, в сегменті супер-комп'ютеріов. За даними на 2011 рік, 413 з 500 провідних супер-комп'ютерів працюють під управлінням Linux. Зміцнюються позиції Linux і в ПК-секторі. Якщо в 1994 році у світі було поставлено 37 млн ПК на базі Linux, то в 2010 році їх відвантаження досягли 351 млн штук.

1.12 Операційна система OS/2

Історія появи, розквіту і практичного відходу зі сцени ОС за загальною назвою OS/2 (перша версія системи – 1987 р.) і дивна і повчальна. Будучи однією з найкращих ОС на ПК, вона проте не стала найпоширенішою, хоча і могла б з легкістю. Головна причина тому – закони бізнесу, а не якість самої ОС. По-перше, IBM не просувала свою систему на ринок ПЗ, орієнтованого на кінцевого користувача, а вирішила продовжувати свою практику роботи з корпоративними клієнтами, ринок яких істотно вужчий. По-друге, основні доходи компанія отримувала від продажу дорогих серверів і іншого устаткування. Для успіху на ринку ОС для ПК необхідно було забезпечити всебічну підтримку своєї ОС навчальною літературою, широкою рекламою тощо. На жаль, цього не сталося, і сьогодні вже мало хто знає про системи OS/2.

Аналітики, що займаються 32-х бітовими операційними системами для персональних комп'ютерів, завжди концентрують свою увагу на «битві» між Microsoft Windows і IBM OS/2, припускаючи, що Microsoft має перевагу. Але не усі згодні з такою точкою зору.

OS/2 v.2.0 була першою доступною і працюючою 32- бітовою операційною системою для персональних комп'ютерів. І вона першою почала черговий круг змагань. Версія OS/2 Warp, призначена для клієнтських машин мереж клієнт-сервер і однорангових мереж, з'явилася на ринку раніше Windows 95. OS/2 Warp була також першою системою, що включила набір засобів підтримки Internet, а також засобів об'єктної орієнтації.

Коли бета-тестери отримали Chicago, першу публічну версію Windows 95, то ті, хто вже використав OS/2, відмітили надзвичайну схожість двох систем. Наприклад, обидві запрошують користувача до роботи за робочим столом; обидві системи розглядають іконки і програми як об'єкти; обидві використовують праву кнопку миші для управління поведінкою об'єктів. Призначений для користувача інтерфейс обох систем має однаковий рівень витонченості. Вимоги до апаратних ресурсів комп'ютера схожі і грунтуються на використанні однакового набору технологій, що лежать в основі системи. Ці технології включають багатозадачність і багатопотчность, здатність виконувати DOS-програми (у основі OS/2 знаходилась DOS) за допомогою віртуальних машин процесорів Intel 80x86, повну 32-х бітову організацію.

І це не випадковість. Через деякий час компанії IBM і Microsoft розійшлися в думці з приводу розвитку і напряму OS/2, оскільки Microsoft хотіла присвятити більше часу ОС Windows, чим OS/2. У 1990 році IBM і Microsoft вирішили працювати незалежно над двома гілками проекту OS/2. IBM взялася контролювати усі подальші версії OS/2 1 і OS/2 2, а Microsoft приступила до розробки OS/2 3. Незабаром, після підписання цієї угоди, компанія Microsoft перейменувала OS/2 3 в Windows NT.

Відколи IBM випустила версію 2.0 OS/2 (1992 р.), а Microsoft вирішила позиціонувати Windows NT як корпоративну ОС самостійно, стала ясно видна прогалина в лінії операційних систем Microsoft, яку і заповнила IBM. Спроби Microsoft висунути Windows 3.1 на ту ж роль розвиненої ОС для настільних систем, що і OS/2, мали обмежений успіх.

У результаті Microsoft стала нести втрати в об'ємах продажів, і, що важливіше, втрачати твердий грунт для своїх операційних систем. Коли стало ясно, що Windows NT навряд чи в повній мірі стане лідером настільних ОС вищого класу, маркетингова машина Microsoft стала менше говорити про можливості Windows NT і почала говорити про можливості Windows 95. Ясно, що IBM і OS/2 мали значний вплив на стратегію Microsoft в області операційних систем.

IBM, у свою чергу, постійно створює здорову конкуренцію для лінії Windows. Windows 95 не порівнянна з OS/2 2.2. Швидше конкурувати почали Windows 95 і **OS/2 Warp 3/0**. Warp – це постріл з далеким прицілом, спрямований на витіснення Windows. І, хоча Warp має деякі початкові переваги і як система виглядає «краще», Windows як і раніше була надійним вибором.

Імена операційних систем можуть змінюватись, але рівновага в битві IBM/Microsoft залишиться тією ж. Існують дві причини – фактична і емоційна, які заважають встановленню перемир'я між цими двома компаніями.

Фактично IBM була в цій області першою. OS/2 перетворилася на працюючий продукт зі своєю версією 2.0 в 1992 році. З того часу вона стала багатозадачною, багатопотоковою системою із зручним об'єктно-орієнтованим інтерфейсом. Зусилля із розвитку OS/2 були неквапливими і постійними, і система отримувала похвали і підтримку на всьому шляху свого розвитку. Проте Windows, як і раніше, тримала найбільшу частку ринку.

Емоційно IBM почуває себе «зрадженою» Microsoft, яка втекла з лав розробників OS/2. Це не зовсім справедливо стосовно до Microsoft, оскільки компанія має право вкладати свої капітали в ту сферу діяльності, яка, на її думку, принесе найбільший прибуток. Хоча Microsoft могла б поводитися тактовніше і продовжувати партнерство з OS/2.

Хоча зараз IBM далеко не та компанія, якою вона була в ті далекі дні, коли вона домінувала на ринку персональних комп'ютерів, їй теж бракує такту. Ця компанія була першою так довго, що вона не вміє виступати на других ролях. Тому з моменту появи версії OS/2 2.0 IBM змінила свою стратегію. Вона стала грати за тими ж правилами, що і інші компанії.

У кінці 1994 року IBM випустила третю головну версію OS/2, яку назвала OS/2 Warp 3 (warp – основа). Зараз ми маємо справу вже з версією OS/2 Warp 4. OS/2 Warp має добре продуманий об'єктно-орієнтований інтерфейс із застосуванням техніки drug-and-drop при виконанні операцій копіювання, видалення, друку, а також деяких інших операцій. Переліки властивостей об'єктів легко доступні в меню, що викликаються клацанням правої клавіші миші. Є спеціальна панель для розміщення часто використовуваних документів або прикладних програм.

До складу OS/2 Warp входить набір утиліт BonusPack, який містить IBM Works – інтегрований програмний пакет початкового рівня. Internet Access Kit, Web Browser і пошта Internet Mail – найповніший набір засобів для мережі Internet з усіх засобів, що поставляються в складі операційних систем. У публікаціях зустрічаються твердження, що він досконаліший, ніж набір для доступу до Internet, реалізований в Windows 95. У лютому 1995 року IBM почала продавати пакет OS/2 Warp 3 Full Pack, який містить бібліотеки Win-OS/2. Ці бібліотеки дають можливість виконувати Windows-програми без придбання ліцензійних копій Microsoft Windows.

Одним з часто критикованих недоліків OS/2 Warp є ті, що вона не підтримує 32-х бітові додатки Windows. Точніше, вона підтримує API Win32s, але не підтримує повний API Win32, який майже повністю підтримує Windows 9x. Ще одним з недоліків OS/2 вважається ті, що вона була спроектована і розроблена без урахування вимог із захисту від несанкціонованого доступу. Це позначається передусім на файловій системі і програмах користувача, які мають можливість боронуй переривань. Отже, сертифікація OS/2 на відповідність якомусь класу захисту(наприклад, класу С2 Помаранчевій книзі, див. підрозділ 3.5 Надійність, захист інформації і безпека) не представляється можливою.

В той же час в OS/2 Warp було недостатньо мережевих функціональних можливостей. Але стан змінився, оскільки влітку 1995 року IBM почала продавати наступну версію OS/2-Warp Connect, яка містила найважливіші драйвери і утиліти. Крім того, Warp Connect надає давно очікувані в OS/2 засоби однорангового мережевого зв'язку. Згідно з повідомленням фірми IBM, в цю версію входило велике число власних драйверів, які могли працювати більш ніж з 70% існуючих адаптерів Ethernet, і більш ніж з 90% адаптерів Token Ring.

Фірма IBM істотно удосконалила свою серверну операційну систему LAN Server версії 4.0, яка може працювати з новою версією OS/2 2.11, що підтримує симетричне мультипроцесування, яке забезпечує його хорошу масштабованість. Версія OS/2 2.11 for SMP перевершила за продуктивністю деякі системи з 3 або 4 процесорами. Так, однопроцесорний OS/2 Warp Server обганяє за продуктивністю двохпроцесорну Windows NT.

Будова OS/2 вважається практично ідеальною і дотопер практично незмінна. Цей факт говорить про глибоку продуманість архітектури системи, адже і до цього дня OS/2 є однією з найпотужніших і найпродуктивніших ОС, особливо серверів.

1.13 Нові досягнення Microsoft Windows

Після грандіозного успіху Windows XP компанія Microsoft випускає Windows Vista (Vista – перспектива) під кодовою назвою Longhorn. Реліз системи відбувся в 2007 році. Нова ОС була свого роду спробою вчинити революцію в оформленні графічного інтерфейсу. Також в Microsoft постаралися усунути недоліки в системі безпеки, що так докучали користувачам XP. ОС Longhorn включала вдосконалений тривимірний інтерфейс користувача, надійнішу систему безпеки і підтримку записуваних універсальних цифрових дисків.

Проте нова система вийшла вкрай посередньою. Про це говорить хоч би те, що ОС зайняла перше місце в конкурсі «Провал року» в 2007 році. Користувачі також були розчаровані в новому продукті від Microsoft. Особливо виділяють проблеми зі швидкодією, несумісністю з багатьма старими програмами, а також завищені системні вимоги, які перевершують заявлені. Після виходу Windows 7 у 2009 році Vista, яка і так не користувалася популярністю, практично повністю «відмерла».

Наступна операційна система компанії Microsoft – Windows 7 – була представлена 22 жовтня 2009 року. Вона повинна була усунути усі недоліки, що були в Vista. Дизайн Aero був доопрацьований, реалізована підтримка старих програм, недоступних для запуску на Windows Vista. Також в Windows 7 з'явився режим Windows XP mode, що дозволяє запускати старі додатки у віртуальній машині Windows XP, чим забезпечується практично повна підтримка старих додатків. Важливою особливістю нової системи є тісніша інтеграція з виробниками драйверів, більшість з яких визначаються автоматично. Нова операційна система припала до смаку великій кількості користувачів. За перші вісім годин кількість попередніх замовлень перевищила попит, який Windows Vista мала за перші 17 тижнів.

Основним недоліком системи є знову ж таки високі системні вимоги, через що автономність ноутбуків в деяких випадках знижувалася до 30%. Незважаючи на це, система залишається популярною. Наприклад, на вересень 2015 року доля Windows 7 займала більше 55% ринку.

У жовтні 2012 року Microsoft представляє черговий продукт – Windows 8. Нова система отримала кардинально новий інтерфейс, більше «заточений» під використання на планшетах. Так, в Windows 8 зникла кнопка «Пуск», на місці якої розташувався доступ до інтерфейсу Metro.

Основними нововведеннями Windows 8, окрім нового інтерфейсу, можна вважати підтримку USB 3.0, вдосконалений пошук і новий диспетчер завдань. Проте велика частина користувачів не оцінила систему. Нова версія Windows 8.1 була спробою виправити недоліки. На своє законне місце повернулася кнопка «Пуск» і стало можливим встановити запуск стандартного робочого столу за умовчанням. Незважаючи на спробу виправити помилки, здійснені в Windows 8, оновлення також було сприйняте без ентузіазму.

Останньою на даний момент ОС Microsoft є Windows 10, яка була представлена в липні 2015 року. Windows 10 повинна об'єднати усі пристрої, включаючи вбудовані системи, смартфони, планшети, ноутбуки, ПК і ігрові консолі. У Windows 10 Microsoft збирає дані про використання комп'ютера. Через це на Microsoft обвалився шквал критики. Windows по праву можна назвати невід'ємною частиною цілого покоління користувачів ПК, вона в будь-якому випадку залишиться актуальною ще не один рік.

1.14 Історія операційних систем Apple

Компанія Apple була заснована 1 квітня 1976 року Стівом Джобсом, Стівом Возняком і інженером компанії Atari Рональдом Уейном (Ronald Wayne). Першим продуктом компанії було дітище Возняка – комп'ютер Apple I. Apple I був заснований на восьмибітовому процесорі 6502 від компанії MOS Technology, який міг ефективно працювати тільки на частоті менше 1 Мгц. Apple I можна було безпосередньо підключати до телевізора за допомогою радіочастотного модулятора, що дозволяло відображати на екрані 24 рядки по 40 символів у кожному. Комп'ютер був запущений в продаж за ціною 666,66 долара. У комплект також входили модулі RAM на 4 КБ і Apple BASIC.

Прошивка Apple I включала System Monitor – програму, яку можна було вважати операційною системою. Розмір програми складав 256 байт, вона використовувла клавіатуру і екран, щоб демонструвати користувачеві командний рядок для перегляду вмісту пам'яті і запуску програм.

Тоді як Apple I протягнув менше року, його наступник Apple II затримався на ринку куди більше і став однією з самих культових машин Apple. Apple II став першим персональним комп'ютером, що підтримував кольорову графіку. У результаті комп'ютер Apple II виявився таким популярним, що згодом вийшло ще декілька його моделей. Для сімейства комп'ютерів Apple II були доступні декілька операційних систем.

Після випуску в 1977 році Apple II усі зрозуміли, що перехід на дисководи був для комп'ютерів життєво важливою задачею. Возняк розробив конструкцію дисковода під назвою Disk II, і разом з цим виникла потреба в дисковій операційній системі (DOS). Перша версія DOS від Apple була випущена в липні 1978 року під назвою Apple DOS 3.1. Ця ОС не мала ніякого відношення до популярної MS-DOS від Microsoft.

У 1980 році був випущений комп'ютер Apple III. Він отримав нову ОС під назвою SOS. Буква S означала «sophisticated» (витончений), хоча спочатку абревіатура розшифровувалася як «Sara' s Operating System» (операційна система Сари), на честь дочки інженера-програміста.

У 1984 році компанією Apple був представлений легендарний комп'ютер Macintosh. Розповідаючи про програмну начинку цих комп'ютерів, було б несправедливо не згадати про те, чим надихалися їх розробники під час творчого процесу. Для цього повернемося в 1968 рік, коли ще не було UNIX, а Apple і Microsoft не існували навіть у проекті.

9 грудня 1968 року в Конференц-центрі Сан-Франциско відбулося технологічне представлення. Інженер Дуглас Енгельбарт (Douglas Engelbart) і його команда з 17 співробітників, що працювали в його науковому центрі при Стенфордскому дослідницькому центрі, продемонстрували систему NLS (oNLine System), над якою вони працювали з 1962 року.

Того ж дня Енгельбарт продемонстрував першу комп'ютерну мишу – трьохкнопковий «вказівний пристрій» з точкою на екрані, що переміщається, і названу «жучком». На нижній стороні корпусу миші були два коліщатка, які могли кататися або ковзати по плоскій поверхні. Кожне коліщатко управляло потенціометром. Користувач рухав мишкою, а відповідні рухи викликали зміни в напрузі, які трансформувалися у відповідні координаи «жучка» на екрані.

Ще однією прибудовою введення даних, яку Енгельбарт продемонстрував у рамках тієї історичної презентації, стала акордна клавіатура – п'ятипальцевим еквівалент повнорозмірної клавіатури. Акордна клавіатура, як пристрій введення даних, мала усього п'ять кнопок. Символи або команди формувалися натисненням декількох клавіш одночасно, як при добуванні акордів на піаніно. Таку клавіатуру можна було тримати однією рукою і набирати будь-які символи. Вона могла використовуватися для введення до 31 текстового символу.

На початку сімдесятих років компанія Xerox задалася ідеєю створення персонального комп'ютера. В результаті вийшов легендарний комп'ютер Alto. Система складалася із вбудованого 16-бітового процесора, растрового графічного екрану з роздільною здатністю 606x808 пікселів, клавіатури з п'ятипальцьовим комплектом клавіш, трьохкнопкової миші (цього разу з одним коліщатком) і корпусу, де знаходився процесор, диски і система електроживлення. Машина оснащувалася портами для підключення принтерів і плотерів, а також портом інтерфейсу ETHERNET із швидкістю 2,94 Мбіт/с для підключення до інших комп'ютерів Alto і лазерних принтерів.

ОС Alto включала драйвери для дисків, клавіатури і монітора, функцію управління пам'яттю, годинником, переривannями та інші функції.

Xerox офіційно представив комп'ютер 8010 STAR Information System на виставці в Чикаго в квітні 1981 року. Апаратне забезпечення STAR було засноване на Alto, але мало удосконалені компоненти (більше пам'яті, більш місткіші диски, збільшена роздільна здатність дисплею).

У січні 1983 року, за рік до виходу Macintosh, компанія Apple випустила свій комп'ютер Lisa, що продавався за захмарною ціною, – 9995 доларів. Цей комп'ютер був оснащений 5-мегагерцовим 32-бітовим процесором Motorola 68k, а його операційна система була схожа на ОС машини Xerox Alto. Доступ в лабораторію Xerox PARC Стіву Джобсу свого часу вдалося отримати в обмін на невеликий пакет акцій Apple.

ОС комп'ютера Lisa, Lisa Office System(OS), мала повністю графічний інтерфейс користувача. В її складі був файловий менеджер з активними іконками, на які можна було натискати курсором миші. У систему було вбудовано додаток для складання динамічних таблиць (LisaCalc) та інструмент для підготовки статичних таблиць (LisaGraph).

Стів Джобс продемонстрував Macintosh 24 січня 1984 року, пізніше відомий як Mac 128K (завдяки 128 кілобайтам вбудованої пам'яті RAM). Він оснащувався 8-мегагерцовим процесором Motorola MC68000. Цей комп'ютер мав вбудований 9-дюймовий монітор з роздільною здатністю 512x342 пікселів і чорно-білою матрицею. Крім того, машина мала один 3-дюймовий дисковод гнучких дисків, що підтримував 400-кілобайтні дискети.

Macintosh працював на розрахованій на одного користувача однозадачній ОС, яка називалася Mac System Software і поміщалася на одній дискеті на 400 Кб. Macintosh ROM містив код низького рівня (для ініціалізації, діагностики апаратного забезпечення, установки драйверів тощо) і «Toolbox» більш високого рівня. Toolbox був колекцією системних програм і загальної бібліотеки, якою могли користуватися програмісти у своїх додатках. Функціонал Toolbox включав управління діалоговими вікнами, шрифтами, іконками, меню, подіями, процесами введення і редагування тексту.

Додаток, який запускався відразу ж після завантаження системи, називався Finder. Він був інтерфейсом для огляду файлової системи і запуску додатків. Однозадачна суть ОС вимагала, щоб користувач закривав працюючий додаток для продовження роботи в Finder. При цьому Macintosh File System (MFS) була плоскою файловою системою: усі файли зберігалися в одній-єдиній директорії. Кожен диск у своєму корені містив теку під назвою «Empty Folder» (порожня тека). Нові теки створювалися шляхом перейменування цієї теки, після чого в директорії з'являлася нова Empty Folder.

Впродовж наступних років після виходу комп'ютера Macintosh компанія Apple займалася удосконаленням його операційної системи, яка була далека від досконалості. За цей період було створено декілька інших систем: System Software Release, System Version, Finder Version, MultiFinder Version, LaserWriter Version та інші.

Одне з найістотніших удосконалень з'явилося, коли Apple зробила можливою спільну багатозадачність за допомогою функції MultiFinder. Ця

функція дозволяла користувачеві одночасно відкривати декілька програм, а також виділяти для цих програм RAM.

Через чотири роки після виходу Macintosh декілька інженерів і менеджерів Apple в березні 1988 року провели збори. На них вони влаштували мозковий штурм, присвячений питанню про подальший розвиток операційній системи Mac розвиватися далі. У ході цього процесу вони записали свої ідеї на картках трьох кольорів: синій, рожевій і червоній.

На синій картці описувався проект з удосконалення існуючої ОС Macintosh. Згодом у процесі розвитку цього проекту буде сформовано ядро System 7.

На рожевій картці був представлений проект революційної операційної системи Apple. Ця система повинна була мати об'єктний характер, повний захист пам'яті, легковагові процеси (потоки), багатозадачність і багато інших сучасних функцій. На червоній картці були викладені ще зухваліші і амбітні ідеї, ніж на рожевій.

System 7, що була розроблена в результаті «синього» проекту, стала найзначимішою ОС Apple. Проте, у світ вона вийшла лише в 1991 році, а до цього Apple встигла випустити ще дві дуже цікаві системи: GS/OS і A/UX.

Apple II виявилася дуже довговічною моделлю. Навіть після виходу Macintosh у 1984 році комп'ютер Apple II як і раніше був присутнім на ринку. У 1986 році був випущений Apple IIGS, що став своєрідним мостом між старим і новим. Це була перша і єдина 16-бітова модель Apple II, що мала вражаючі мультимедійні можливості. Букви «GS» означали графіку (graphics) і звук (sound). Операційна система Apple ProDOS стала доступна в 8- і 16-бітовій версіях, щоб підтримувати Apple IIGS.

У кінці 1988 року Apple представила ОС A/UX – власну версію Unix, сумісну з POSIX. Ранній варіант A/UX був заснований на 4.2BSD і AT&T UNIX System V Release 2. ОС A/UX включала такі функції як контроль робіт, сигнали, мережу (AppleTalk, STREAMS, TCP/IP, сокети, NFS з YP тощо), файлову систему Berkeley (ffs).

Цікаво, що в A/UX багато функцій операційної системи Macintosh було взято з ОС Unix. ОС A/UX 2.x використала System 6, тоді як A/UX 3.x була комбінацією вищезгаданого середовища Unix і System 7. Файлова система A/UX виглядала як іконка з дисководом в System 7 Finder. Був можливий одночасний запуск додатка Macintosh, додатка Unix (командний рядок і X Window) і навіть додатків DOS.

Коли в 1991 році вийшла ОС System 7, вона була велетенським стрибком вперед порівняно з попередніми версіями операційних систем для Macintosh. До числа основних її функцій входили такі:

1. Вбудований MultiFinder.
2. Вбудовані мережеві служби і служби обміну файлами.
3. Підтримка роботи з 32-бітовою пам'яттю.
4. Використання віртуальної пам'яті.

Перший комп'ютер Macintosh з блоком управління пам'яттю (MMU) був випущений у 1987 році, і називався Macintosh II. System 7 стала першою операційною системою, яка скористалася його перевагами. Macintosh II став першим комп'ютером Macintosh, що підтримував кольорову графіку. Завдяки використанню кольорової карти Apple, Mac II міг відображати 8-бітовий колір з роздільною здатністю 640x480 пікселів і підтримувати 256 кольорів.

З виходом ОС System 7 уперше з'явилися такі технології як AppleScript (макромова на системному рівні для автоматизації завдань), ColorSync (система управління кольором), PowerTalk (програма для спільної роботи і використання електронної пошти), QuickTime (мультимедійний додаток для перегляду, копіювання і вставки відео, анімацій, зображень і аудіофайлів), TrueType (шрифтова технологія) і WorldScript (підтримка декількох мов на рівні системи). Проте, з виходом першої версії System 7 ці додатки не були інтегровані в систему.

В цей же час Apple створила альянс з компаніями IBM і Motorola, у рамках якого в планах з розвитку апаратного забезпечення Apple з'явилися процесори PowerPC. Для реалізації цієї задачі були потрібні фундаментальні зміни в структурі операційної системи Macintosh.

У 1991 році Apple, IBM і Motorola об'єднали зусилля і створили альянс AIM Alliance, метою якого була розробка спільної апаратної платформи CHRP. У результаті співпраці була створена архітектура PowerPC. Першим процесором PowerPC став 601, який був спроектований як 64-бітова архітектура, яка повинна була динамічно перемикатися між 64- і 32-бітовим режимами. У комп'ютерах Apple процесор PowerPC проіснував достатньо довго.

ОС System 7.1.2 була першою системою, що підтримувала PowerPC. Для управління PowerPC використовувалося наноядро (ядро, що своїм розміром поступалося навіть мікроядру), яке працювало в захищеному режимі.

Через декілька років, у 1996 році, Apple представила продукт під назвою Apple Network Server, який прожив дуже недовге життя. Цей сервер на базі PowerPC мав такі особливості структури, як відсіки приводів з можливістю гарячої заміни (з використанням RAID), системи живлення і вентилятори з можливістю гарячої заміни, підтримку підключення додаткових SCSI-пристроїв і PCI плат.

Network Server оснащувався операційною системою AIX для серверів Apple Network Servers, заснованою на AIX від IBM, і не підтримував Mac OS. В ОС AIX з'явилися такі функції як захист пам'яті, випереджаючий мультипроцесінг, багатопотоковость, підтримка різних мережевих протоколів тощо. Користувач мав можливість вибрати роботи з командним рядком або двома варіантами графічного інтерфейсу: AIXwindows або Common Desktop Environment(CDE).

Лінійка Network Server була знята з виробництва в 1997 році, коли в компанію повернувся Стів Джобс. У той час Apple знаходилася на межі банкрутства, і було прийнято рішення зосередити свою увагу на мінімальній кількості продуктів. Серверні рішення до їх числа не входили.

По мірі того, як Apple на початку дев'яностих продовжувала здавати свої позиції, пальму першості перехопила Microsoft. ОС Windows 3.x, випущена в 1993 році, мала величезний успіх. Наступне покоління системи, що проходило під кодовою назвою «Chicago», спочатку було заплановане до випуску в тому ж році. Проте, розробка системи затягнулася, і в результаті вона вийшла тільки через два роки під назвою Windows 95. 1993 рік був ознаменований ще одним знаковим релізом – Windows NT. Це була система з ширшим функціоналом і підвищеною продуктивністю, розрахована на потужніші професійні машини і сервери.

Apple необхідно було прореагувати на домагання Microsoft. Новий проект, під назвою Star Trek, який реалізовувався спільно з компанією Novell, був спробою портировать (адаптувати) Mac OS на процесори x86. На початку 1994 року компанія Apple оголосила, що планує використати десятирічний досвід розробок для створення своєї нової операційної системи Mac OS 8, яка проходила під кодовою назвою «Copland». Реалізовуючи цей проект, компанія планувала досягти декількох своїх цілей.

1. Адаптувати RISC в якості ключової технології, зробивши систему повністю натівною (рідною) для PowerPC.
2. Інтегрувати, удосконалити і зміцнити такі наявні технології як OpenDOC QuickDraw GX (графічна архітектура для введення тексту, графіки, кольору і друку), ColorSync, QuickDraw 3D, а також інструменти спільної роботи: PowerTalk і PowerShare.
3. Зберегти і удосконалити простоту використання інтерфейсу Mac OS, зробивши цю систему сумісною з розрахованим на багато користувачів режимом.
4. Розширити сумісність з такими платформами як DOS і Windows.
5. Зробити системи Mac OS кращими мережевими клієнтами.

На початку 1990-х робота над Copland прискорилася, і до середини останнього десятиліття минулого століття інженери Apple чекали дива, яке перетворить компанію. Проте, процес розробки постійно пробуксовував. За першу половину дев'яностих років встигло вийти декілька версій комплекту розробки драйверів (DDK), проте плани з випуску системи в 1996 році представлялися усе більш нереальними.

У травні 1996 року Apple нарешті приймає рішення про відмову від реалізації проекту Copland. Було заявлено, що найвдаліші компоненти Copland будуть включені в майбутні релізи наявної операційної системи. Було вирішено почати готуватися до випуску System 7.6, яка на момент релізу була офіційно перейменована в Mac OS 7.6.

В якийсь момент Apple розглядала можливість співпраці з корпорацією Microsoft у створенні операційної системи, заснованої на Windows NT. Серед інших можливих варіантів розглядався продукт Solaris від Sun Microsystems і BeOS від компанії Be. Насправді, Apple мала велике бажання купити Be. Компанія Be була заснована колишнім главою відділу Apple з розробки

продуктів Жаном-Луї Гассе (Jean-Louis Gassée), який очолював сильну команду, що розробила вражаючу операційну систему. BeOS мала все те, чого так пристрасно бажала Apple: захист пам'яті, пріоритетну багатозадачність, симетричний мультипроцесінг і навіть могла працювати на PowerPC (а згодом і на x86). BeOS була спроектована так, щоб ефективно обробляти мультимедіа. В той же час BeOS була ще незавершеним і неперевіреним продуктом.

Гассе був обізнаний про те, як сильно Apple хоче одержати Be, і тому виставив за свою компанію ціну більше 500 мільйонів доларів. Загальний об'єм інвестицій в Be складав всього 20 мільйонів доларів, а Apple оцінювала цю компанію в 50 мільйонів. В процесі переговорів Apple погодилася збільшити ціну до 125 мільйонів доларів, тоді як Гассе погодився понизити свою ціну до 300 мільйонів. Зрештою Apple виступила з пропозицією про придбання компанії за 200 мільйонів доларів. Проте, жадність узяла верх, і глава Be запропонував Apple «остаточну» ціну в 275 мільйонів доларів і був у повній упевненості, що Apple погодиться. Проте, цього так і не сталося: угода зірвалася.

Ще однією потенційною мішенню Apple у той час була нова компанія Стіва Джобса NeXT, чия операційна система, на відміну від BeOS, була вже готова і доступна на ринку. Попри те, що NeXT у той час ще не могла похвалитися приголомшливими успіхами, ОС OPENSTEP зустріла дуже теплий прийом на ринку. Джобс активно пропонував Apple скористатися своєю технологією і запевняв, що вона у своєму розвитку випередила конкурентів на декілька років.

У результаті Apple уклала угоду з NeXT. У лютому 1997 року компанія придбала NeTX за ціною більше 400 мільйонів доларів. Хоча ця ціна була значно вища за ту, що Apple могла б заплатити за Be, зрештою це придбання виявилося для компанії доленосним і радикально змінило в кращу сторону її долю. І ось через 12 років, придбавши NeXT, компанія Apple повернула у свої ряди свого «блудного» директора Джобса. Джобсу вистачило одного року, щоб повністю реструктурувати діяльність компанії і змусити її рости. Багато в чому це пов'язано з новими і дуже успішними продуктами, які почали з'являтися після повернення Джобса.

31 травня 1985 року Стіва Джобса «попросили» з компанії Apple, і йому довелося починати все спочатку. Зрештою він прийняв рішення заснувати нову компанію разом ще з п'ятьма співробітниками Apple, які пішли разом з ним. Його план полягав у створенні ідеального комп'ютера для університетів, коледжів і лабораторій, за допомогою якого можна проводити складні наукові дослідження.

Рада директорів Apple була дуже розсерджена, коли дізналася, що Джобс переманив у свій стартап п'ять працівників своєї колишньої компанії. Керівництво Apple в зв'язку з цією ситуацією навіть подало на Стіва до суду, проте менш ніж через рік позов було вирішено відкликати. Звичайно, йшлося про компанію під назвою NeXT Computer, Inc.

Початок епохи NeXT був досить вражаючим з інвестиційної точки зору. Джобс вклав в цю компанію 7 мільйонів доларів власних грошей. Компанія Canon вирішила інвестувати в компанію NeXT ще 100 мільйонів доларів. Компанія NeXT прагнула створити комп'ютер, який був би ідеальний як на

вигляд, так і з функціональності. Його материнська плата мала дуже практичну конструкцію і привабливий дизайн, а магнієвий корпус системного блоку кубічної форми був у чорному забарвленні і прикрашався матовою обробкою.

Джобс представив комп'ютер NeXT 12 жовтня 1988 року. Його ОС називалася NEXTSTEP і використовувла у своєму ядрі порт CMU Mach 2.0 (з середовищем 4.3BSD). Віконний сервер цієї системи був заснований на Display Postscript – альянсі мови опису сторінок і технологій віконних систем. Порт Mach, використаний в NEXTSTEP, також включав декілька конкретних функцій NeXT і функцій наступних поколінь CMU Mach.

На момент випуску кубічного комп'ютера NeXT операційна система NEXTSTEP була доступна у версії 0.8. Повноцінний реліз 1.0 був випущений через ще один рік. Через рік після виходу версії 1.0 стала доступна операційна система NEXTSTEP 2.0, що пропонувала такі удосконалення як підтримка приводів CD-ROM і кольорових моніторів, перевірка орфографії в реальному часі, драйвери пристроїв, що динамічно завантажуються.

На виставці NeXTWORLD Expo 1992 року була представлена ОС NEXTSTEP 486 для машин на базі x86a. Ціна цієї версії складала 995 доларів США. Остання версія NEXTSTEP, що проходила під номером 3.3, була випущена в лютому 1995 року і включала такі потужні інструменти для розробки програмних додатків як Project Builder, Interface Builder і багато інших. Ці інструменти були доповнені величезними бібліотеками інтерфейсів користувача, баз даних, розподілених об'єктів, мультимедіа, мережевих функцій тощо.

Незважаючи на всі переваги NEXTSTEP і елегантність апаратного забезпечення, NeXT виявився економічно нерентабельним, унаслідок непідйомної ціни комп'ютера. На початку 1993 року Стів Джобс оголосив, що компанія виходить з бізнесу з виробництва пристроїв і зосереджує всю свою увагу на продовженні розробки ОС NEXTSTEP.

Разом з операційною системою NeXT було розроблено ядро Mach, яке згодом послужило фундаментом для операційних систем Apple.

Mach – мікроядро операційної системи, розроблене в Carnegie Mellon University при проведенні дослідницьких робіт в області операційних систем для розподілених і паралельних обчислень. Це один з найперших прикладів мікроядра, але й досі він є стандартом для інших подібних проектів.

Коли була розроблена система Mach, UNIX вже встигла відмітити свій 15-річний ювілей. Хоча розробники Mach погоджувалися з тим, що UNIX є дуже значимою і дуже корисною платформою, в той же час вони відмічали, що UNIX вже не відрізняється минулою легкістю і простотою налаштування. В цілому мета створення Mach полягала в реагуванні на складність, що неухильно збільшувалася, і заплутаність UNIX. Зокрема, при розробці системи розробники поставили перед собою задачу добитися такого:

1. Повна підтримка мультипроцесінгу.
2. Використання інших функцій сучасної апаратної архітектури, яка з'явилася в той час.

3. Скорочення кількості функцій ядра, що повинне було зробити його простішим. В той же час функції повинні були мати досить загальний характер для того, щоб дозволити розробляти на базі Mach декілька операційних систем.
4. Повна сумісність з UNIX.
5. Виправлені недоліки попередніх систем.

У 1986 році система Mach називалася «Новим ядром-фундаментом для розробки на базі UNIX». І хоча далеко не всі так вважали, Mach зрештою стала досить успішною операційною системою. Пізніше провідний розробник проекту Mach Річард Рашид згадував, що після серії невдалих спроб дати своєму дітищу ім'я він вирішив назвати систему MUCK (Multiprocessor Universal Communication Kernel – мультипроцесорне універсальне комунікаційне ядро). А один з його колег, італієць Даріо Джузе (Dario Giuse), ненавмисно вимовляв MUCK як «Mach». Цей варіант сподобався розробникам більше, і вирішили залишити його.

Версія Mach 2.0, а також ще успішніша версія 2.5 мали монолітні системи: BSD і Mach розташовувалися в одному і тому ж адресному просторі. Проект з розробки Mach 3 стартував в університеті Carnegie Mellon і згодом був продовжений в OSF (Open Software Foundation – Фонд відкритого програмного забезпечення). Це була перша версія зі «справжнім мікроядром» в тому сенсі, що BSD запускався в якості простору користувача задачі Mach, причому тільки фундаментальні функції виконувалися ядром Mach. Багато операційних систем були перенесені на концептуальну «віртуальну машину», створену на базі Mach API. Крім того, декілька операційних систем режиму користувача мали можливість виконувати команди на базі Mach.

Apple і OSF почали проект з розробки порту Linux, призначеного для різних платформ Power Macintosh, на яких використовувалася версія Mach, створена OSF. В результаті розробки було створено ядро під назвою «osfmk», а ОС, що вийшла, отримала найменування MkLinux. Перша версія цього «Linux на базі Mach», заснованого на Mach і Linux 1.3.x, вийшла у світ на початку 1996 року під назвою MkLinux DR1. У новіших версіях системи вже використовувалася платформа Linux 2.0.x. Mac OS X як свою основу використовує osfmk (а частиною ядра при цьому є BSD) і включає багато функцій, що з'явилися в системі з виходом MkLinux. Організація OSF незабаром була перейменована в Open Group, а потім – в Silicomp.

Стратегія Apple у сфері операційних систем після придбання NeXT носила подвійний характер. З одного боку компанія планувала продовжити удосконалювати Mac OS для ринку споживчих комп'ютерів, а з іншого – створювати нову високопродуктивну ОС під назвою Rhapsody, засновану на технології від NeXT. ОС Rhapsody була розрахована на ринок серверних машин і корпоративний сегмент.

Першою ОС, що вийшла після купівлі NeXT, стала ОС версії 7.6. Після цього планувалося випустити версію 7.7, яка згодом була перейменована в Mac OS 8.0.

Mac OS 8 отримала багатопотоковий додаток Finder, який дозволяв одночасно запускати декілька файлових операцій, підтримував запуск контекстного меню при натисненні Control і кліка миші, персональний веб-хостинг, а також відрізнявся важливими вдосконаленнями в області управління електроживленням. У комплект ОС входили браузери Microsoft Internet Explorer і Netscape Navigator. Версія ОС Mac 8.5 була розрахована на використання тільки на машинах, які були оснащені процесором PowerPC.

Mac OS 9 вийшла у світ у 1999 і позиціонувалася Apple як «краща у світі операційна система для Інтернету». Пов'язано це було, головним чином, з тим, що ця система стала першою версією Mac OS, яку можна оновлювати через Інтернет. У систему входили різні корисні функції безпеки, такі як шифрування файлів і механізм Keychain для зберігання паролів.

Ще одним важливим компонентом була інсталяція Carbon API, на яку в той час доводилося приблизно 70 відсотків від усіх Mac OS API. Цей набір інструментів розробки пропонував сумісність з Mac OS 8.1 і пізнішими системами. Останнім релізом «старої» Mac OS (названої «Classic») стала версія 9.2.2, яка вийшла в кінці 2001 року.

У 1997 році на Всесвітній конференції розробників WWDC була уперше продемонстрована операційна система Rhapsody, яка складалася з таких основних компонентів:

1. Ядро і відповідні підсистеми, засновані на Mach і BSD.
2. Реалізація розширеного OpenStep API під назвою Yellow Box.
3. Віртуальна машина Java.
4. Сумісна з Mac OS підсистема, названа Blue Box.
5. Призначений для користувача інтерфейс у дусі Mac OS з деякими функціями OPENSTEP.

Після Rhapsody DR2 компанія Apple знову змінить свою стратегію розвитку ОС, проте цього разу, нарешті, вирішить рухатися в бік створення «нової» системи. Щоб досягти цієї мети, компанії знадобиться цілих три роки.

До виходу версії Rhapsody DR3, Apple в березні 1999 року представила систему Mac OS X Server 1.0, яка розглядалася як удосконалена версія Rhapsody. Система мала інтегровані WebObjects, стрим-сервер QuickTime, веб-сервер Apache, засоби для завантаження і адміністрування через мережу.

Для Mac OS X було випущено чотири попередні версії системи Developer Preview, призначені для розробників і названі DP1-4. У вересні 2000 року вийшла бета-версія нової системи (Mac OS X Public Beta), яка пропонувалася за ціною 29,95 долара. В цій бета-версії з'явилося декілька важливих технологій Apple, які компанія уперше реалізувала, і яких не було у версіях DP.

24 березня 2001 року була випущена Mac OS X 10.0. Незабаром був переглянутий план розвитку Mac OS X Server так, щоб він був синхронізований з розвитком клієнтської системи. Відтоді зародилася тенденція, яка полягала в тому, щоб спочатку випускалася клієнтська версія, а незабаром після неї – серверна. Наступні великі релізи Mac OS X, що встигли вийти до цього моменту, наведені в таблиці 1.1.

Таблиця 1.1 – Релізи Mac OS X 10.x

Версія	Кодове позначення	Дата випуску
10.0	Cheetah	24 березня 2001 р.
10.1	Puma	29 вересня 2001 р.
10.2	Jaguar	24 серпня 2002 р.
10.3	Panther	24 жовтня 2003 р.
10.4	Tiger	29 квітня 2005 р.
10.5	Leopard	26 жовтня 2007 р.
10.6	Snow Leopard	28 серпня 2009 р.
10.7	Lion	20 липня 2011 р.
10.8	Mountain Lion	25 липня 2012
10.9	Mavericks	22 жовтня 2013
10.10	Yosemite	16 жовтня 2014
10.11	El Capitan	30 вересня 2015
10.12	Sierra	20 вересня 2016
10.12.5	Sierra	15 травня 2017

На відміну від попередниць, Mac OS X є повноцінною, сертифікованою UNIX'03 операційною системою. Це означає, що більшість програм, написаних для BSD, Linux і інших UNIX-подібних систем скомпілюються і працюватимуть на Mac OS X без додаткових змін у коді.

Версія Mac OS X для PowerPC залишається сумісною із старими Mac OS додатками через емуляцію так званої Classic, яка дозволяє користувачам запускати Mac OS 9 як процес в Mac OS X. Classic не підтримує комп'ютери на процесорах Intel. Незабаром Mac OS X була портирована на iPhone і iPod touch.

Mac OS X – друга за популярністю платформа у світі, її ринкова доля у вересні 2011 року складала 6,03 %. З цього числа 3,46 % доводилося на версію Mac OS 10.6, 1,17 % – на Mac OS 10.5, 1,03 % – на Mac OS 10.7 і ще 0,34 % – на частку Mac OS 10.4.

Згідно зі статистикою використання операційних систем, ОС Mac OS X на грудень 2016 року займала четверте місце в списку найпопулярніших ОС у світі. Як і у випадку з Windows, загальна доля Mac падає під натиском мобільних платформ. У грудні 2016 року система займала усього лише 4,9% ринку проти 6,5% п'ятьма роками раніше (у дужках вказана ринкова доля на десктопах):

12.2011 – 6,5% (7,0%); 12.2012 – 6,3% (7,7%); 12.2013 – 5,7% (7,8%);
12.2014 – 5,3% (8,7%); 12.2015 – 5,5% (9,8%); 12.2016 – 4,9% (11,0%).

На даний момент Mac OS X має власний красивий інтерфейс Aqua. Вона проста у використанні і доброзичлива. У ній використовується середовище програмування Core Foundation, що включає такі компоненти як Carbon API, Cocoa API і Java API. Графічне середовище представлене використанням таких технологій як QuickTime, Quartz Extreme і OpenGL. Важливою перевагою Mac OS X є її безпека при роботі в інтернеті, вона непогано захищена від інтернет-атак, та і кількість вірусів здатних її уразити на сьогодні нікчемно мало.

1.15 Операційні системи для мобільних пристроїв

Мобільні пристрої (mobile intelligent devices – мобільні телефони, комунікатори) – це пристрої, якими зараз користуються постійно для голосового зв'язку, а також для запису або обробки інформації або для виходу в Інтернет.

У перше десятиліття після своєї появи більшість смартфонів працювали під управлінням Symbian OS. Цю операційну систему обрали такі популярні бренди, як Samsung, Sony Ericsson, Motorola і Nokia. Але частку ринку Symbian почали відбирати інші операційні системи, наприклад RIM Blackberry OS і Apple iOS (випущена для першого iPhone в 2007 році).

Епоха домінування операційних систем типу Symbian на ринку мобільних телефонів, мабуть, закінчується, і вони поступаються місцем сучаснішим і таким, що забезпечують кращий призначений для користувача інтерфейс ОС Google Android і Microsoft Windows Phone [6].

1.15.1 Історія ОС Google Android

На момент написання цього посібника платформа Google Android вже є помітним явищем в області програмного забезпечення для мобільних пристроїв. Новою платформою зацікавилися провідні світові виробники мобільної електроніки і стільникові оператори, а багато хто з них вже випускає великий асортимент мобільних пристроїв, що працюють під управлінням Android.

У чому ж полягає унікальність платформи Android? Основна ідея Google полягає в тому, що компанія пропонує у відкритий доступ початкові коди своєї операційної системи. Для розробки Google надає набір зручних інструментів і добре документований комплект SDK. Це повинно з часом призвести до появи великої кількості програмного забезпечення для цієї платформи.

За декілька років Android став найуспішнішим проектом для мобільних телефонів. Android захоплює ринок мобільних телефонів, поступово витісняючи з нього загальновизнаних лідерів. Google Android встановлюється тепер не лише на смартфони, ця платформа підходить і для планшетів і нетбуків.

Великим кроком у розвитку Google Android стало відкриття в жовтні 2008 року онлайн-магазину додатків – Android Market, в якому можна придбати програми і інший софтвер для пристроїв на базі нової платформи. Крім того, тепер для розробників програмного забезпечення з'явилася можливість брати плату за свої додатки в Android Market, що робить розробку додатків під цю платформу ще привабливішою.

На сьогодні Android – це вже не просто операційна система для смартфону, а ціла інфраструктура. На «зеленому роботові» працюють телефони, планшети, телевізори, розумний годинник та інші гаджети, а скоро і автомобілі управлятимуться за допомогою Android.

Остання версія Android має порядковий номер 5 і кодову назву Lollipop. Система отримала значні оновлення в дизайні, функціональності. Проте про новий 5.0 ми розповімо окремо, а почати хочеться все ж ще з тих часів, коли проект Android навіть не належав Google.

Багато хто вважає, що історія Android почалася в 2008 році, коли була випущена перша версія Android 1.0. Але насправді все закрутилося на 5 років раніше. У 2003 році Енді Рубін з товаришами (Нік Сірс, Кріс Уайт і Рич Майнер) вирішив створити мобільну операційну систему і зареєстрували компанію Android Inc. Розробники спочатку зосередилися на пристроях, які могли б постійно знаходитися у користувачів, визначати місце розташування GPS і автоматично підлаштовуватися під потреби людини.

Так і сталося, що до 2005 року Енді і друзі витратили всі свої засоби на розробку системи, але завдяки щасливому випадку до них придивилися з Google і 17 серпня 2005 року корпорація стала повноправним власником маленької Android Inc.

До випуску в 2008 році готується перше складання Android 1.0. Проте на початку 2007 року у Google немає партнера, який випустив би телефон на новій ОС. У Google вибирають між LG і HTC. Корейській LG цікавий ринок США, проте вона боїться співпраці з невідомим партнером і використовує домовленості з Google тільки для того, щоб укласти контракти з Microsoft по створенню смартфонів з Windows Mobile. А ось HTC була готова до спільної роботи, та до того ж тайванська компанія могла швидко створювати робочі зразки. Перша робоча версія датується 15.05.2007 і називалася вона тоді M3. І тільки до серпня 2008 року в Google розробили версію 0.9, щоб представити версію ОС 1.0 у вересні 2008 року. З 22 жовтня 2008 року, оператор T-Mobile в США починає продажу HTC Dream (T-Mobile G1), першого Android- смартфону. Інтерес до HTC Dream в США був величезний, оператор продав до 23 квітня 2009 року 1 мільйон пристроїв.

У 2010 році Google випустила ще одну версію Android, нову 2.2 Froyo, в якій збільшилась швидкодія додатків, які використовували JIT-компіляцію (англ. *Just – in - time compilation*, компіляція «на льоту»), і з'явилася підтримка Adobe Flash. На початку 2011 року з'являється перша спроектована спеціально для планшетних ПК версія Android – 3.0 Honeycomb. Восени 2011 року Google випускає версію Android 4.0 Ice Cream Sandwich, яка стає першою кросплатформеною версією для смартфонів і планшетів. Саме з версії 4.0 Android почав набувати звичні контури і нормальну функціональність.

У 2012-2013 роках нічого особливого з Android не сталося після глобальних змін з об'єднанням планшетної і смартфонної версій. У 2013 році на ринку з'являється Nexus 5, знову ж таки, в результаті співпраці з LG. І для нього і інших пристроїв виходить нова версія Android 4.4 KitKat. З виходом KitKat спростився доступ до сервісу Google Now. Тепер його виклик уніфікований – досить лише провести по екрану пальцем зліва направо. Раніше способи доступу до Google Now варіювалися залежно від моделі смартфону (натиснення на кнопку Home, поштовхи тощо).

Таким чином, Android KitKat став практично здійсненою версією системи. Робота усіх сервісів була відлагоджена, зовнішній вигляд наздогнав Apple з інтуїтивності і стилю. Загалом, це була вже хороша допрацьована система. Але Google хотілося більшого. Потрібний був Material Design, Android Wear, робота з автомобілями і багато чого іншого.

У червні 2014 року на конференції Google I/O для розробників була представлена нова версія Android L. Точна назва системи невідома, але ясно, що порядковий індекс у неї буде 5. І ось, лише в жовтні 2014 року Google признається, що буква L – це Lollipop, а два нові пристрої на цю ОС – це Nexus 6 (Motorola) і Nexus 9 (HTC). Найочевидніші зміни в Android 5 включають новий призначений для користувача інтерфейс, названий авторами «матеріальний дизайн». Основними принципами цього дизайну є тіні, інформація, показана в оформленні, що нагадує про шари паперу, яскравого, але в той же час з більшою інформативністю. Удосконалюються і повідомлення, які тепер доступні з екрану блокування, і з будь-якого додатку у вехній частині екрану. До кінця 2014 року більше 53% проданих на ринку смартфонів були на Android- платформі, тоді як на Apple iOs – 41,6% [7].

На конференції Google I/O 2016 була представлена нова версія ОС Android 7 – Nougat, фінальна версія якої була випущена 22 серпня 2016 року. Нарешті, тестування Android 8.0, яке почалося в березні 2017 року, в серпні 2017 року підійшло до кінця. Як і очікувалося, ОС отримала назву Oreo (Android O), на честь печива.

Операційна система Android 8.0 має цілий ряд поліпшень. Так, Android Oreo зводить до мінімуму будь-яку активність у фоновому режимі, що украй позитивно позначається на енергоспоживанні. Режим «Картинка в картинці» тепер з'явиться на усіх пристроях. Є можливість змінювати розміри кожного вікна. У режимі «Не турбувати» у користувача з'явилася можливість відключати на 15, 30 або 60 хвилин сповіщення для будь-якого конкретного додатку. Крім того, нова мобільна ОС завантажується в два рази швидше і працює з більш високою швидкістю.

На сьогодні (серпень 2017 р.) випущені 14 версій системи Android.

1.15.2 Історія ОС Windows Phone

Історія цієї мобільної операційної системи нерозривно пов'язана як з Windows настільною, так і з її портативними версіями. Її навіть можна назвати результатом багаторічних спроб Microsoft зайняти нішу ОС для портативних пристроїв, і цього разу спроба вийшла вдалою. Втім, прямих попередників цієї ОС теж не можна назвати невдахами. Вони досить успішно освоювали ринок довгі роки, і Windows Phone можна назвати природним еволюційним розвитком мобільної операційної системи від Microsoft.

Все починалося із вбудовуваної ОС Windows CE, яка використовується в промисловості й досі, а також з Windows Mobile, чий розвиток закінчився в кінці 2000-х років версією 6.5.3. В основу шостої Windows Mobile лягла споріднена Windows CE 5.2, ставши фактично її дуже сильно удосконаленою версією [8].

Можна згадати HTC Touch – він буквально підірвав ринок ранніх смартфонів, ставши першим з тих, що управлявся пальцем, а не стилусом, і він базувався якраз на ОС Windows Mobile шостій версії. Не кажучи вже про численні КПК, які тоді випускалися на тій же Windows Mobile.

У 2008 році була представлена нова версія цієї ОС –Windows Mobile 6.1. Під її управлінням вийшов смартфон, що теж став згодом легендарним, – Samsung WiTu. Він отримав нестандартний широкоформатний екран, який в результаті і прижився, а також знамениту оболонку TouchWiz.

Ще рік потому була представлена остання версія Windows Mobile 6.5 з істотним оновленням функціональності, де був оптимізований інтерфейс для управління пальцями. Тоді ж з'явився перший і єдиний смартфон на цій ОС з мультитачем (від англ. *Multi-touch* – «множинне торкання»): HTC HD2, на якому можна було запускати Android, Windows Phone 7, Ubuntu і навіть Windows RT.

І, нарешті, в 2010 році світ побачив Windows Phone 7. Windows Phone 7 повністю втратила сумісність з попередніми версіями і отримала абсолютно новий «плитковий» інтерфейс Metro UI, повноцінний магазин додатків і багато чого іншого. З'явився повноцінний офісний пакет Microsoft Office Mobile, мобільний браузер Internet Explorer та інші корисні речі. У лютому 2011 році на конференції MWC 2011 з'явилися Internet Explorer Mobile 9 і підтримка HTML5, підтримка фронтальної камери, 19 нових мов, багатозадачність у сторонніх додатках, хмара SkyDrive.

Нова версія (Windows Phone 8), яку можна назвати основною поточною версією 8.1, вийшла 29 жовтня 2012 року. Це було абсолютно інше, нове покоління ОС, засноване на архітектурі Windows NT, не сумісній з попереднім поколінням. Нова версія отримала ідеологію призначеного для користувача інтерфейсу із попередньої версії, але з'явилися новий екран блокування, живі плитки, нова версія Internet Explorer (десята).

У кінці 2012 року побачила світ остання версія «сімки» з індексом 7.8. Ця версія стала останньою в лінійці Windows Phone 7. Для старих смартфонів впровадження цієї версії продовжилося до літа 2013 року.

Нарешті, 2 квітня 2014 року пройшла презентація найсучаснішої версії Windows Phone – 8.1. Найпомітніше оновлення – голосовий помічник Cortana, аналог Siri у iOS і Google Now в Android. Cortana спочатку була доступна тільки англійською мовою для регіону США, версія в Росії для російської мови з'явилася в четвертому кварталі 2014 року.

Серед інших оновлень – перейменування хмарного сервісу SkyDrive в OneDrive, удосконалення роботи багатозадачності, підтримка управління жестами, розділ швидких налаштувань у верхньому випадному меню (по аналогії з Android 5, але набагато простіше і зручніше), роздільне регулювання гучності для дзвінків/повідомлень і додатків, підтримка Bluetooth 4.0LE і двох SIM- карт.

Найактуальніша новина 2010 року – вихід операційної системи Windows 10. Ця ОС буде єдиною для усіх гаджетів, від смартфону до десктопа. Тобто, нинішнє розділення Windows на Windows і Phone піде в минуле.

Формально нову ОС назвали Windows 10 Mobile, і її пріоритет в плані синхронізації – розширений набір контенту, що синхронізується, і функцій, універсальні додатки (для мобільної і настільної версій), використання смартфона як ПК з підтримкою миші і клавіатури. Мінімальні вимоги для системи аналогічні як для Windows Phone 8, включаючи оперативну пам'ять і частоту процесора.

1.16 ОС для хмарних обчислень

Хмарні обчислення (cloud computing) є одним з найпопулярніших напрямів розвитку ІТ. «***Хмара»*** (cloud) – це представлення сервісів, що надаються через Інтернет або іншу комунікаційну мережу. ***Хмарні обчислення*** – модель обчислень, заснована на динамічно масштабованих і віртуалізованих ресурсах (даних, додатках, ОС та ін.), які доступні і використовуються як сервіси через Інтернет і реалізуються за допомогою високопродуктивних центрів обробки даних.

З точки зору користувачів існує сукупність «хмар» (загальнодоступні, корпоративних, приватних та ін.), що надаються різними компаніями, для використання потужних обчислювальних ресурсів, яких немає в індивідуального користувача. Як правило, «хмарні» сервіси платні.

Головною перевагою використання хмарних обчислень, яка покладена в основу технології, є балансування робочого навантаження, за рахунок чого досягається більш ефективне використання ресурсів обчислювальної системи.

Недолік хмарних обчислень в тому, що користувач виявляється повністю залежним від використовуваної їм «хмари» (у якій доступні використовувані ним дані і програми) і не може управляти не лише роботою «хмарних» комп'ютерів, але навіть резервним копіюванням своїх даних. У зв'язку з цим виникає ціла низка важливих запитань щодо безпеки хмарних обчислень, збереження конфіденційності призначених для користувача даних тощо. Далеко не усі з цих питань на сьогодні розв'язані.

Jolicloud представляє Joli OS, яку можна скачати і встановити на системі, або використати її через браузер. Joli OS пропонує повноцінний досвід взаємодії з хмарними системами. Joli OS може зберігати і використати ваші додатки в будь-який момент. Jolicloud має більше 15000 веб-додатків, якими можна скористатися, використовуючи цю систему.

Glide OS – це відмінний вибір для групової роботи над проектом. Команда Glide пропонує 30Гб безкоштовного дискового простору. ОС, можливо, не найпривабливіша зовні, але дуже зручна.

SilveOS є хмарною ОС, розробленою за допомогою Silverlight. Її можна запустити в будь-якому браузері на пристрої, де встановлений Silverlight. Тут є певна кількість вбудованих додатків, які дозволяють писати повідомлення, слухати музику, робити замітки. Можна також встановлювати Silverlight-додатки з Інтернету в SilveOS.

Найпопулярніша «хмарна» платформа – Microsoft Windows Azure (***хмарна ОС***) і Microsoft Azure Services Platform (реалізована на основі Microsoft.NET). Windows Azure можна розглядати як ***«ОС в хмарі».*** Користувачеві немає необхідності турбуватися про її інсталяцію на його комп'ютері, який може не мати для цього необхідних ресурсів. Все, що вимагається, це мати Web-браузер для запуску і використання через браузер хмарних сервісів.

Нині усі великі компанії (Microsoft, IBM, HP, Dell, Oracle та ін.) розробляють свої системи хмарних обчислень. Є тенденція до інтеграції цих корпоративних систем в єдину доступну користувачеві «хмару».

1.17 Операційні системи СРСР

Вітчизняні розробники, майже нічого не знаючи про аналогічні роботи американських колег, створювали свої оригінальні системи, у тому числі – ОС. Наприклад, ідея багатопоточності (multi-threading) була реалізована в ОС «Ельбрус» [1; 3] ще в кінці 1970-х років, а в популярних зарубіжних ОС (UNIX, Solaris, Windows NT) багатопоточність з'явилася тільки в кінці 1980-х – початку 1990-х років. Серед передових оригінальних розробок в області комп'ютерної апаратури і ОС 1960-х – 1970-х рр. слід виділити передусім ЕОМ БЕСМ-6, її операційні системи ОС ДИСПАК, ОС ДІАПАК і її системне і прикладне програмне забезпечення [3; 4].

Розробником БЕСМ-6 був Інститут точної механіки і обчислювальної техніки АН СРСР під керівництвом академіка Сергія Олексійовича Лебедєва, засновника усієї вітчизняної обчислювальної техніки.

У Всесоюзному науково-дослідному інституті приладобудування (ВНДІП, 1969-1974, нині Російський федеральний ядерний центр – ВНДІТФ, Всесоюзний науково-дослідний інститут технічної фізики ім. акад. Б.І. Забабахіна, – під керівництвом В. Ф. Тюріна було розроблено ряд операційних систем для ЕОМ М-20, М-220 (В.Ф. Тюрін, В. С. Авраменко) і БЕСМ-6 (ОС ДИСПАК, ОС ДІАПАК) [4; 5].

Розробниками операційної системи ДИСПАК (диспетчер пакетної обробки завдань) для сімейства машин БЕСМ-6, в якій уперше в СРСР була реалізована робота з магнітними дисками, були В. Ф. Тюрін, Н. І. Шулепов, Ю. В. Озорнін, Л. В. Шинкарьова, С. А. Зельдінова, І. Д. Бокова, В. І. Зуєв. Три розробники ОС були удостоєні Державної премії.

Плід праці системних програмістів вийшов настільки вдалим, надійним, стабільним в роботі і при цьому простим в експлуатації, що завод-виробник вирішив використовувати ОС ДИСПАК в якості основної серійної операційної системи для всього Радянського Союзу.

Через чотири роки на базі ОС ДИСПАК була створена нова версія ОС ДІАПАК (діалогово-пакетний режим роботи), яка дозволяла автоматизувати практично всі дії оператора. Основний внесок у створення системи ДІАПАК внесли В. Ф. Тюрін, Ю. В. Озорнін, Н. І. Шулепов, С. А. Зельдінова, Л. В. Шинкарьова, В. К. Корякін, Г. П. Охріменко, В. С. Авраменко. Робота була виконана на рівні світових стандартів системного програмування.

До кожної БЕСМ-6 були підключені десятки терміналів, які працювали під управлінням діалогових систем КРАБ, ДЖИН, ПРИМУС та ін. Це при обсязі оперативної пам'яті БЕСМ-6 всього в 32 сторінки по 4096 байт і швидкодії до 1 млн операцій в секунду. Роботу БЕСМ-6 і її ОС відрізняла висока надійність.

Іншою передовою розробкою 1970-х – 1980-х років був багатопроцесорний обчислювальний комплекс (БОК) «Ельбрус-1» і «Ельбрус-2» [1; 3]. Ідейним натхненником проекту «Ельбрус» став сам С. А. Лебедєв, потім ним керували академік В. С. Бурцев, а після нього – член-кор. АН СРСР Б. А. Бабаян.

Комерційним прототипом «Ельбрусу» була відома серія комп'ютерів фірми Burroughs (США): B5000/B5500/B6700/B7700 [2]. Однак розробникам

«Ельбрусу» і його операційної системи вдалося запропонувати і реалізувати цілий ряд власних оригінальних ідей і методів.

У 1980 році Тюрін В. Ф., Зельдінова С. А. з інституту прикладної математики (ІПМ) перейшли в інститут точної механіки і обчислювальної техніки (ІТМіОТ) в проект «Ельбрус». Вони стали відповідальними за проведення випробувань ОС ДИСПАК для БОК ЕЛЬБРУС1-К2. Процесор БОК ЕЛЬБРУС1-К2 був сумісний з БЕСМ-6 за командами для користувача.

У 1985 році почалися роботи по створенню обчислювального комплексу (ОК) ЕЛЬБРУС-Б. Розробка системного ПЗ ОК ЕЛЬБРУС-Б була виконана В. Ф. Тюріним, С. А. Зельдіновой, Н.Є. Балакиревим. М. Г. Чайковський розробив макроассемблер, ФОРТРАН для ОК ЕЛЬБРУС-Б. Державні випробування ОК ЕЛЬБРУС-Б і ОС ДИСПАК успішно пройшли в 1988 році. У 1991 році за створення системного програмного забезпечення для ОК ЕЛЬБРУС-Б колективу розробників (В. Ф. Тюрін, С. А. Зельдінова, Н.Є. Балакірєв, М. Г. Чайковський) була присуджена премія Ради міністрів СРСР [4].

Однак на початку 1970-х років у розвитку вітчизняної обчислювальної техніки і її системного програмного забезпечення почався новий, несподіваний для більшості користувачів і фахівців, етап. Уряд СРСР ухвалив безпрецедентне рішення про створення в якості основної на досить довгий період часу (як спочатку планувалося, на 20-30 років) вітчизняної серії – Єдиної Системи ЕОМ (ЄС ЕОМ) – шляхом копіювання американських комп'ютерів серії IBM 360. Відповідно, усе базове системне програмне забезпечення, в тому числі і ОС, також було адаптовано до використання в СРСР [5].

Це рішення викликало великі проблеми з фінансуванням у розробників вітчизняних архітектур ЕОМ. Це також викликало великі труднощі в користувачів і розробників програмного забезпечення, так як далеко не всі добре володіли англійською мовою.

Контрольні питання і тести до розділу 1

Контрольні питання

1. Дайте найбільш повне визначення операційної системи.
2. Назовіть основні задачі системи пакетної обробки.
3. У чому полягала головна ідея при використанні спеціальної програми, відомої під назвою монітор (диспетчер)?
4. В якому поколінні ОС з'явилися такі поняття, як «спулінг» («підкачування») і «багатозадачність»?
5. Як називався варіант багатозадачної системи, при якому в кожного користувача був свій термінал?
6. В яких ОС був уперше реалізований режим розподілу часу?
7. В якій ОС уперше почали застосовувати віртуальну пам'ять?
8. Дайте визначення віртуальній пам'яті.
9. Назовіть основні задачі операційних систем реального часу.
10. Назовіть основні відмінності перших мережевих операційних систем від багатотермінальних ОС.

11. Яка знаменна подія на початку 80-х років пов'язана з історією операційних систем?
12. Хто є автором першої дискововой операційної системи CP/M?
13. На базі якої ОС була створена видозмінена система MS-DOS?
14. Назвіть ім'я винахідника графічного інтерфейсу користувача.
15. На базі якої ОС була розроблена ОС Linux?
16. Назовіть основні відмінності між Linux і іншими ОС.
17. В якій ОС уперше була застосована мікроядерна архітектура операційної системи?
18. В якому році з'явилася перша версія ОС під загальною назвою OS/2?
19. Чому користувачі без ентузіазму сприйняли ОС Windows 8?
20. Які пристрої повинна об'єднати ОС Windows 10?
21. Під управлінням якої ОС в перше десятиліття після своєї появи працювали більшість смартфонів?
22. У чому ж полягає унікальність платформи ОС Android, яка за пару років стала найуспішнішим проектом для мобільних телефонів?

Тести

1. Як була названа перша мобільна ОС, розроблена в 1970 році К. Томпсоном, Б. Керніганом і Д. Рітчі?
   1) Linux;
   2) MULTICS;
   3) UNIX;
   4) MS DOS.
2. Вкажіть перше сімейство програмно-сумісних машин, що працювали під управлінням однієї і тієї ж операційної системи:
   1) IBM PC;
   2) IBM/360;
   3) PowerPC;
   4) PDP-11.
3. Як і в якому середовищі запускалася перша версія Windows?
   1) як графічна оболонка, командою win в системі MS-DOS;
   2) як самостійна операційна система на процесорах Intel;
   3) як частину ОС MacOS;
   4) як діалект UNIX.
4. Яка матеріальна база комп'ютерних систем першого покоління?
   1) механічні пристрої;
   2) напівпровідникові елементи;
   3) інтегральні мікросхеми;
   4) електронні лампи.
5. Хто є творцем операційної системи Linux?
   1) Пол Аллен;
   2) Ендрю Таненбаум;
   3) Кен Томпсон;
   4) Лінус Торвальдс.

6. Можливість інтерактивної взаємодії користувача і програми виникла з появою:

1) систем пакетної обробки;
2) систем розподілу часу;
3) мультипрограмних обчислювальних систем;
4) мережевих ОС.

7. Яка технічна база характерна для першого періоду обчислювальної техніки (1945-1955 років)?
   1) напівпровідникова;
   2) інтегральні мікросхеми;
   3) лампи.
8. Що було прообразом сучасних ОС?
   1) системи пакетної обробки;
   2) бібліотеки математичних і службових програм;
   3) компілятори з символічних мов;
   4) системи розподілу часу.
9. Планування завдань стало можливим:
   1) з появою систем пакетної обробки;
   2) з появою попереднього запису пакету завдань на магнітну стрічку;
   3) з появою попереднього запису пакету завдань на магнітний диск.
10. Чому операційна система OS/2 не має права претендувати на відповідність класу захисту C2?
    1) тому що в цій ОС неможливо виконати ізоляцію програмних модулів за допомогою механізмів захисту пам'яті;
    2) тому що програми користувача мають право заборони переривань;
    3) тому що в ній не забезпечений спільний доступ до файлів.
11. У 1968 році Дуглас Енгельбарт продемонстрував першу трьохкнопкову комп'ютерну мишу і перший графічний інтерфейс користувача. В якій ОС були вперше застосовані ці нововведення?
    1) в ОС Alto для комп'ютера Alto (компанія Xerox);
    2) в ОС Lisa для комп'ютера Lisa (компанія Apple, 1983 р.);
    3) в ОС Mac System Software для комп'ютера Macintosh (компанія Apple, 1984 р.).
12. Posix – це:
    1) модуль ядра ОС Unix, працюючий в режимі користувача;
    2) назва ОС;
    3) сукупність стандартів, які використовуються в ОС Unix;
    4) назва архітектури обчислювальної машини.

2 ПРИЗНАЧЕННЯ І КЛАСИФІКАЦІЯ ОС

Сучасний комп'ютер складається з одного або декількох процесорів, оперативної пам'яті, дисків, принтера, клавіатури, миші, дисплея, мережевих інтерфейсів та інших різноманітних пристроїв введення-виведення. У результаті виходить досить складна система. Якщо кожному програмісту, що створює прикладну програму, потрібно буде розбиратися у всіх тонкощах роботи всіх цих пристроїв, то він не напише жодного рядка коду. Більш того, управління всіма цими компонентами і їх оптимальне використання є дуже непростим завданням. З цієї причини комп'ютери оснащені спеціальним рівнем програмного забезпечення, який називається ***операційною системою.***

Під операційною системою розуміють комплекс управляючих програмних засобів, які, з одного боку, забезпечують інтерфейс між апаратурою комп'ютера і користувачем з його задачами, а з іншого боку – призначені для ефективного використання ресурсів обчислювальної системи і організації надійних обчислень.

ОС виконує функції управління обчислювальними процесами, розподіляє ресурси обчислювальної системи і створює програмне середовище, в якому виконуються програми користувача. Таке середовище називають ***операційним***.

Наприклад, в далекі 50-і роки при розробці перших систем програмування, передусім, створювали програмні модулі для підсистеми введення-виведення, а вже потім – для операцій, що часто зустрічаються, і функцій. Завдяки цьому програмісти могли просто звертатися до функцій введення-виведення та інших функцій і процедур, що позбавляло їх від створення цих програмних компонентів «з нуля», і від необхідності знати усі подробиці роботи контролерів введення-виведення і відповідних інтерфейсів.

Наступний крок в автоматизації створення готових до виконання програм полягав у тому, що транслятор з алгоритмічної мови більш високого рівня вже сам міг підставляти замість операторів типу READ або WRITE усі необхідні виклики до бібліотечних програмних модулів. Зрештою виникла ситуація, коли при створенні програм програмісти могли взагалі не знати багатьох деталей управління конкретними ресурсами обчислювальної системи, а повинні тільки звертатися до деякої програмної підсистеми з відповідними запитами і отримувати від неї необхідні функції і ***сервіси.*** Ця програмна підсистема і є ОС, а набір її функцій, сервісів і правила звернення до них якраз і утворюють те базове поняття, яке ми називаємо ***операційним середовищем***.

Паралельне існування термінів «операційна система» і «операційне середовище» викликано тим, що ОС у загальному випадку може підтримувати декілька операційних середовищ. Наприклад, ОС OS/2 Warp може виконувати свої 32-розрядні програми, 16-розрядні програми OS/2 1-го покоління, а також 16-розрядні програми в операційному середовищі MS-DOS і Windows.

Можна сказати, що операційне середовище – це те системне програмне оточення, в якому можуть виконуватися програми, створені за правилами роботи цього середовища.

2.1 Різноманітність операційних систем

Історія операційних систем налічує вже понад півстоліття. За цей час було розроблено величезну кількість різноманітних операційних систем, але не всі вони отримали широку популярність. У цьому розділі ми наведемо короткий огляд декількох різних операційних систем.

Операційні системи мейнфреймів. До вищої категорії відносяться операційні системи мейнфреймів (великих універсальних машин) – комп'ютерів, що займають цілі зали і до сих пір ще можна зустріти у великих центрах обробки корпоративних даних. Такі комп'ютери відрізняються від персональних комп'ютерів обсягами введення-виведення даних. Мейнфрейми, мають тисячі дисків і петабайт даних, – це звичайне явище, які знаходить своє застосування в якості потужних веб-серверів, серверів великих інтернет-магазинів і серверів, що займаються міжкорпоративними транзакціями [9].

Операційні системи мейнфреймів орієнтовані переважно на одночасну обробку певної кількості завдань, більшість з яких вимагає колосальних обсягів введення-виведення даних. Зазвичай вони пропонують три види обслуговування: пакетну обробку, обробку транзакцій і роботу в режимі розподілу часу.

Пакетна обробка – це одна з систем обробки стандартних завдань без участі користувачів. У них немає запитів користувачів, вхідні дані збираються заздалегідь для подальшої обробки. Вхідні дані збираються і обробляються в партіях (пакетах), звідси і назва пакетної обробки.

Системи обробки транзакцій справляються з великою кількістю дрібних запитів, наприклад обробкою чеків у банках або бронюванням авіаквитків. Кожна елементарна операція невелика за обсягом, але система може справлятися з сотнями і тисячами операцій в секунду.

Робота в ***режимі розподілу часу*** дає можливість віддаленим користувачів одночасно запускати на комп'ютері свої завдання, наприклад, запити до великої бази даних. Всі ці функції тісно пов'язані один з одним, і часто операційні системи універсальних машин виконують їх у комплексі. Прикладом операційної системи універсальних машин може послужити OS/390.

Мережеві та розподілені операційні системи. Трохи нижче за рівнем стоять мережеві операційні системи, які працюють на серверах, і представлені дуже потужними персональними комп'ютерами, робочими станціями або навіть універсальними машинами. Вони одночасно обслуговують багато користувачів, забезпечуючи їм загальний доступ до апаратних і програмних ресурсів.

Комп'ютерна мережа – це набір комп'ютерів, пов'язаних комунікаційною системою і забезпечених відповідним програмним забезпеченням, що дозволяє користувачам мережі отримувати доступ до ресурсів цього набору. При організації мережевої роботи операційна система відіграє роль інтерфейсу, екрануючого від користувача всі деталі низькорівневих програмно-апаратних засобів мережі. Залежно від того, який віртуальний образ реальної апаратури комп'ютерної мережі створює ОС, розрізняють мережеві і розподілені ОС.

Мережева ОС надає користувачеві якусь віртуальну систему, яка не повністю приховує розподілену природу реального прототипу. Під мережевий

ОС розуміють операційну систему окремого комп'ютера, що включає засоби для роботи в мережі.

Користувач мережевий ОС завжди знає, що він має справу з мережевими ресурсами і що для доступу до них потрібно виконати деякі операції. Також повинен знати, де зберігаються його файли, і використати наявні команди для їх переміщення, а також знати, на якій машині виконується його завдання. В ідеальному випадку ОС повинна надавати користувачеві мережеві ресурси так, як якщо б вони були ресурсами єдиної централізованої віртуальної машини. Це – магістральний напрям розвитку ОС. Така операційна система носить назву розподіленої ОС.

Розподілена ОС існує як єдина операційна система в рамках обчислювальної системи і змушує набір мережевих машин працювати як віртуальний унікальний процесор. Кожен комп'ютер мережі виконує частину функцій цієї єдиної ОС. Користувач в такому випадку, взагалі кажучи, не має відомостей про те, на якій машині виконується його робота.

Мережеві засоби підрозділяються на три компоненти:

- ***серверна частина ОС*** – засоби надання локальних ресурсів і послуг у загальне користування;
- ***клієнтська частина ОС*** – засоби запиту доступу до віддалених ресурсів і послуг, які належать іншим комп'ютерам мережі;
- ***транспортні****,* або ***комунікаційні засоби ОС*** **–** засоби, які спільно з комунікаційною системою забезпечують обмін повідомленнями в мережі.

Сервери можуть надавати послуги друку, зберігання файлів або веб-служб. Інтернет-провайдери для обслуговування своїх клієнтів використовують відразу декілька серверних машин. При обслуговуванні веб-сайтів сервери зберігають веб-сторінки і обробляють запити, що надходять.

Типовими представниками серверних операційних систем є різні варіанти ОС Unix (HP-UX компанії Hewlett-Packard, Solaris компанії Sun, FreeBSD та інші), Linux і ОС Windows Server 201x, починаючи з NT.

Багатопроцесорні операційні системи. Зараз все ширше використовується об'єднання багатьох центральних процесорів в єдину систему, що дозволяє домогтися великої обчислювальної потужності. Залежно від того, як саме відбувається це об'єднання, а також які ресурси загального користування об'єднуються, ці системи називаються паралельними комп'ютерами, мультикомп'ютерами або багатопроцесорними системами. Їм потрібні спеціальні операційні системи, в якості яких часто застосовуються особливі версії серверних операційних систем, оснащені спеціальними функціями зв'язку, сполучення і синхронізації.

З появою багатоядерних процесорів для персональних комп'ютерів і ноутбуків операційні системи стали працювати щонайменше з невеликою багатопроцесорною системою. Згодом, схоже, число ядер буде тільки рости. За роки попередніх досліджень були накопичені великі знання про багатопроцесорні операційні системи, і використання цього арсеналу в багатоядерних системах не повинно викликати особливих ускладнень. На

багатопроцесорних системах можуть працювати багато популярних операційних системи, включаючи Windows і Linux.

Операційні системи персональних комп'ютерів. Завданням операційних систем персональних комп'ютерів є якісна підтримка роботи окремого користувача. Всі їх сучасні представники підтримують багатозадачність. При цьому вже в процесі завантаження на одночасне виконання запускаються десятки програм. Типовими прикладами можуть служити операційні системи Linux, FreeBSD, Windows 7-10 і OS X компанії Apple.

Операційні системи кишенькових персональних комп'ютерів. Для планшетів, смартфонів та інших кишенькових комп'ютерів розроблені свої операційні системи – мобільні операційні системи. Мобільна ОС управляє мобільним пристроєм, її дизайн підтримує бездротовий зв'язок і мобільні додатки.

Кишенькові персональні комп'ютери, спочатку відомі як КПК, або PDA (Personal Digital Assistant – персональний цифровий секретар), являють собою невеликі комп'ютери, які під час роботи тримають в руці. Найвідомішими їх представниками є смартфони і планшети. Більшість таких пристроїв начинені багатоядерними процесорами, GPS, камерами та іншими датчиками, достатнім обсягом пам'яті і складними операційними системами. На цьому ринку домінують операційні системи Android від Google і iOS від Apple, але у них є багато конкурентів.

Вбудовані операційні системи. Вбудовані системи працюють на комп'ютерах, які керують різними пристроями. Вони компактні і ефективні, і здатні працювати з обмеженим числом ресурсів. Оскільки на цих системах установка для користувача програм не передбачається, їх зазвичай комп'ютерами не вважають. Прикладами пристроїв, де встановлюються вбудовані комп'ютери, можуть послужити мікрохвильові печі, телевізори, автомобілі, холодильники, звичайні телефони і MP3-плеєри. Найбільш популярними в цій області вважаються операційні системи Embedded Linux, QNX і VxWorks.

Операційні системи реального часу. Ще один різновид операційних систем – це системи реального часу. Ці системи характеризуються тим, що час для них є ключовим параметром.

Головним об'єктом операційних систем реального часу є їхня швидка і передбачувана реакція на події. Система управляється подіями, перемикається між завданнями на основі їх пріоритетів, з розподілом часу перемикання завдань.

Наприклад, в системах управління виробничими процесами комп'ютери, що працюють в режимі реального часу, повинні збирати відомості про процес і використовувати їх для керування верстатами на підприємстві. Досить часто вони повинні відповідати дуже жорстким тимчасовим вимогам.

Якщо операція повинна бути проведена точно в строк (або в певний період часу), то ми маємо справу з ***системою жорсткого реального часу***. Багато подібних систем зустрічається при управлінні виробничими процесами, в авіаційно-космічному електронному обладнанні, у військовій та інших подібних областях застосування. Ці системи повинні давати абсолютні гарантії того, що певні дії будуть здійснюватися в конкретний момент часу.

Іншим різновидом подібних систем є ***система м'якого реального часу***, в якій хоча і небажано, але цілком допустимо недотримання терміну будь-які дії, що завдає непоправної шкоди. До цієї категорії відносяться цифрові аудіо- або мультимедійні системи. Смартфони також є системами м'якого реального часу.

Категорії операційних систем для КПК, вбудованих систем і систем реального часу в значній мірі перекриваються один з одним за властивими їм ознаками. Практично всі вони мають принаймні деякі аспекти систем м'якого реального часу. Вбудовані системи і системи реального часу працюють тільки з тим програмним забезпеченням, яке заклали в них розробники цих систем. Його користувачі не можуть додати в цей арсенал власне програмне забезпечення, що істотно полегшує вирішення задач захисту.

2.2 Призначення і функції операційної системи

Сьогодні існує велика кількість різних типів операційних систем, що відрізняються сферами застосування, апаратними платформами і методами реалізації. Природно, це обумовлює і значні функціональні відмінності цих ОС. Навіть у конкретної операційної системи набір виконуваних функцій частенько визначити не так просто. Та функція, яка сьогодні виконується зовнішнім по відношенню до ОС компонентом, завтра може стати її невід'ємною частиною і навпаки.

Призначення ОС можна розкласти на такі складові.

1. **Зручність**. ОС робить використання комп'ютера простими і зручними.
2. **Ефективність.** ОС дозволяє ефективно використати ресурси комп'ютера.
3. **Можливість розвитку.** ОС має бути організована так, щоб вона дозволяла ефективну розробку, тестування і впровадження нових програм-додатків і системних функцій, причому це не повинно заважати нормальному функціонуванню ОС.

Кінцевий користувач може бути не знайомий з деталями апаратного забезпечення комп'ютера. Комп'ютер для нього є набором апаратних засобів і системного програмного забезпечення. Програму-додаток можна створити на будь-якій системі програмування.

Створення програми-додатка в машинних кодах, яка повністю відповідає за управління апаратним забезпеченням комп'ютера, є досить складним завданням. Для спрощення процедури створення додатків існує набір системних програм, у тому числі утиліти. Останні реалізують функції ОС, які часто використовуються для роботи з файлами і пристроями введення-виведення. Програміст використовує ці засоби для розробки власних програм, а вони під час свого виконання звертаються до утиліт для виконання певних функцій.

2.2.1 Основні функції операційних систем

Оскільки ОС виступає в ролі посередника, що полегшує програмістам і програмам-додаткам доступ до різних можливостей комп'ютерної системи, то при вивченні операційних систем дуже важливо з усього різноманіття виділити

ті функції, які властиві всім операційним системам як класу продуктів. Список таких можливостей, які надаються ОС, можна представити таким чином.

1. Розробка програм. Сприяє програмістові при розробці програм, ОС надає йому різноманітні інструменти і сервіси. Ці сервіси реалізовані у вигляді спеціальних системних програм (системи компіляції, відладки тощо) і утиліт (обслуговування файлів і тому подібне), які підтримуються операційною системою, але не входять в її ядро. Ці програми є інструментальними засобами для розробки додатків.

2. Виконання програм. Для запуску програм необхідно виконати ряд дій. Потрібно завантажити в основну пам'ять коди команд і дані, ініціювати зовнішні пристрої і файли, а також підготувати інші ресурси. ОС виконує цю рутинну роботу замість користувача.

3. Доступ до пристроїв введення-виведення. Для управління роботою кожного пристрою введення-виведення потрібний свій особливий набір команд або сигналів управління. ОС надає користувачеві стандартний інтерфейс, який приховує усі ці деталі і забезпечує програмістові доступ до пристроїв введення-виведення за допомогою простих команд читання і запису.

4. Контрольований доступ до файлів. При роботі з файлами з боку ОС передбачаються не лише глибокі знання і розуміння роботи пристроїв введення-виведення (диски, CD і тому подібне), але і знання структур даних, які записані у файлах. Багатозадачні ОС, крім того, можуть забезпечувати роботу механізмів захисту при зверненні до файлів.

5. Системний доступ. ОС керує доступом до спільно доступної обчислювальної системи в цілому, а також до окремих її ресурсів. Вона може забезпечити захист ресурсів і даних від несанкціонованого звернення до них, а також розв'язувати конфліктні ситуації (у багатозадачних ОС це можуть бути драйвери віртуальних пристроїв).

6. Знаходження помилок і їх обробка. При роботі комп'ютерної системи можуть виникати різноманітні збої, до яких можна віднести як внутрішні, так і зовнішні, що виникають в апаратурі. Можливі також програмні помилки (арифметичне переповнювання, спроба звернення до елементу пам'яті, доступ до якої заборонений). Реакція ОС на помилку може бути різною (заміщення сегменту або сторінки пам'яті, просте повідомлення про помилку, аварійне завершення програми).

7. Облік використання ресурсів. Сучасна ОС повинна мати засоби обліку різних ресурсів і відображення параметрів продуктивності обчислювальної системи. Така інформація є украй важливою, особливо у зв'язку з підвищенням її пропускної спроможності.

Таким чином, можна зробити висновок, що операційна система комп'ютера є комплексом взаємозв'язаних програм, який діє як інтерфейс між додатками і користувачами з одного боку, і апаратурою комп'ютера з іншого боку. Відповідно до цього визначення ОС виконує в основному дві групи функцій:

- надання користувачеві або програмістові замість реальної апаратури комп'ютера розширеної віртуальної машини, з якою зручніше працювати і яку легше програмувати;

– підвищення ефективності використання комп'ютера шляхом раціонального управління його ресурсами відповідно до деякого критерію.

Розглянемо деякі з цих функцій детальніше.

2.2.2 ОС як віртуальна машина

Для того щоб успішно розв'язувати свої задачі, сучасний користувач або навіть прикладний програміст може обійтися без досконального знання апаратних пристроїв комп'ютера. Йому не обов'язково знати, як функціонують різні електронні блоки і електромеханічні вузли комп'ютера. Більше того, дуже часто користувач може не знати навіть системи команд процесора. Користувач-програміст звик мати справу з потужними високорівневими функціями, які йому надає операційна система (рис. 2.1).

Так, наприклад, при роботі з диском програмістові, що пише додаток для роботи під управлінням ОС, або кінцевому користувачеві ОС досить представляти його у вигляді деякого набору файлів, кожен з яких має ім'я. Послідовність дій при роботі з файлом полягає в його відкритті, виконанні однієї або декількох операцій читання або запису, а потім в закритті файлу.

Рисунок 2.1 – ОС як віртуальна машина

Такі деталі, як використовувана при записі частотна модуляція або поточний стан двигуна механізму переміщення магнітних головок читання/запису, не повинні хвилювати програміста. Саме операційна система приховує від програміста велику частину особливостей апаратури і надає можливість простої і зручної роботи з необхідними файлами.

Якби програміст працював безпосередньо з апаратурою комп'ютера, без участі ОС, то для організації читання блоку даних з диска програмістові довелося б використати більше десятка команд з вказівкою певної кількості параметрів: номера блоку на диску, номера сектора на доріжці і т. п. А після завершення операції обміну з диском він повинен був би передбачити у своїй програмі аналіз результату виконаної операції.

Враховуючи, що контролер диска здатний розпізнавати більше двадцяти різних варіантів завершення операції, можна вважати програмування обміну з диском на рівні апаратури не найтривіальнішим завданням. Не менш

обтяжливою виглядає і робота користувача, якби йому для читання файлу з терміналу потрібно було задавати числові адреси доріжок і секторів.

Операційна система позбавляє програмістів не лише від необхідності безпосередньо працювати з апаратурою дискового накопичувача, надаючи їм простий файловий інтерфейс, але і бере на себе усі інші рутинні операції, пов'язані з управлінням іншим апаратним обладнанням комп'ютера: фізичною пам'яттю, таймерами, принтерами тощо.

У результаті реальна машина, що здатна виконувати тільки невеликий набір елементарних дій, які визначаються її системою команд, перетворюється на віртуальну машину, що виконує широкий набір набагато потужніших функцій. Віртуальна машина теж управляється командами, але це вже команди іншого, вищого рівня: видалити файл з певним ім'ям, запустити на виконання деяку прикладну програму, підвищити пріоритет завдання, вивести текст з файлу на друк. Таким чином, призначення ОС полягає в наданні користувачеві/програмістові деякої розширеної віртуальної машини, яку легше програмувати і з якою легше працювати, чим безпосередньо з апаратурою, що становить реальний комп'ютер.

2.2.3 ОС як диспетчер ресурсів

Операційна система не лише надає користувачам і програмістам зручний інтерфейс до апаратних засобів комп'ютера, але і є механізмом, що розподіляє ресурси комп'ютера.

До числа основних ресурсів сучасних обчислювальних систем можуть бути віднесені такі ресурси: процесори, основна пам'ять, таймери, набори даних, диски, принтери, мережеві пристрої і деякі інші. Ресурси розподіляються між процесами. Процес (завдання) є базовим поняттям більшості сучасних ОС і часто коротко визначається як програма в стадії виконання. Програма – це статичний об'єкт, що є файлом з кодами і даними. Процес – це динамічний об'єкт, який виникає в операційній системі після того, як користувач або сама операційна система вирішує «запустити програму на виконання», тобто створити нову одиницю обчислювальної роботи.

У багатьох сучасних ОС для позначення мінімальної одиниці роботи ОС використовують термін «потік», або «нитка», при цьому змінюється суть терміну «процес». Детальніше це ми розглянемо пізніше. В інших розділах ми будемо дотримуватися спрощеного тлумачення, відповідно до якого для позначення програми, що виконуєтся, тільки термін «процес».

Управління ресурсами обчислювальної системи з метою найефективнішого їх використання є основним призначенням операційної системи. Наприклад, мультипрограмна операційна система організовує одночасне виконання відразу декількох процесів на одному комп'ютері, по черзі перемикаючи процесор з одного процесу на інший, виключаючи простої процесора, що викликаються зверненнями процесів до введення-виведення. ОС також відстежує і розв'язує конфлікти, що виникають при зверненні декількох процесів до одного і того ж пристрою введення-виведення або до одних і тих же

даних. Критерій ефективності, відповідно до якого ОС організовує управління ресурсами комп'ютера, може бути різним. Наприклад, в одних системах важливий такий критерій, як пропускна спроможність обчислювальної системи, в інших – час її реакції. Відповідно до вибраного критерію ефективності операційні системи по-різному організовують обчислювальний процес.

Управління ресурсами включає розв'язання наступних загальних, не залежних від типу ресурсу, задач:

- планування ресурсу – тобто визначення, якому процесу, коли і в якій кількості (якщо ресурс може виділятися частинами) слід виділити цей ресурс;
- задоволення запитів на ресурси;
- відстежування стану і облік використання ресурсу, тобто підтримка оперативної інформації про те, зайнятий чи вільний ресурс, і яка доля ресурсу вже розподілена;
- розв'язання конфліктів між процесами.

Для розв'язання цих загальних задач управління ресурсами різні ОС використовують різні алгоритми, особливості яких і визначають вигляд ОС в цілому, включаючи характеристики продуктивності, сферу застосування і навіть призначений для користувача інтерфейс. Наприклад, вживаний алгоритм управління процесором значною мірою визначає чи може ОС використовуватися як система розподілу часу, система пакетної обробки або система реального часу.

Задача організації ефективного спільного використання ресурсів декількома процесами є дуже складне, і ця складність породжується в основному випадковим характером виникнення запитів на споживання ресурсів. У мультипрограмній системі утворюються черги заявок від одночасно виконуваних програм до розподілених ресурсів комп'ютера: процесора, сторінки пам'яті, до принтера, до диска. Операційна система організовує обслуговування цих черг за різними алгоритмами: в порядку вступу, на основі пріоритетів, кругового обслуговування тощо.

Таким чином, управління ресурсами складає важливу частину функцій будь-якої операційної системи, особливо мультипрограмної. Більшість функцій управління ресурсами виконуються операційною системою автоматично і прикладному програмістові недоступні.

2.2.4 Управління процесами

Найважливішою частиною операційної системи, що безпосередньо впливає на функціонування обчислювальної машини, є підсистема управління процесами.

Для кожного новостворюваного процесу ОС генерує системні інформаційні структури, які містять дані про потреби процесу в ресурсах обчислювальної системи, а також про фактично виділені йому ресурси.

Щоб виконати процес, операційна система повинна призначити йому область оперативної пам'яті, в якій будуть розміщені коди і дані процесу, а також

надати йому необхідну кількість процесорного часу. Крім того, процесу може знадобитися доступ до таких ресурсів, як файли і пристрої введення-виведення.

В інформаційні структури процесу часто включаються допоміжні дані, що характеризують історію перебування процесу в системі. Наприклад, яку частку часу процес витратив на операції введення-виведення, а яку на обчислення, його поточний стан (активний або заблокований), міра привілейованості процесу (значення пріоритету). Дані такого роду можуть враховуватися операційною системою при ухваленні рішення про надання ресурсів процесу.

У мультипрограмній операційній системі одночасно може існувати декілька процесів. Частина процесів породжується за ініціативою користувачів та їх додатків. Такі процеси називають призначеними для користувача. Інші процеси, що називаються системними, ініціалізувалися самою операційною системою для виконання своїх функцій.

Оскільки процеси часто одночасно претендують на одні і ті ж ресурси, то в обов'язки ОС входить підтримка черг заявок процесів на ресурси, наприклад, черги до процесора, до принтера, до послідовного порту.

Важливим завданням операційної системи є захист ресурсів, що виділені цьому процесу, від інших процесів. Одним з ресурсів процесу, що найретельніше захищаються, є області оперативної пам'яті, в якій зберігаються коди і дані процесу. Сукупність усіх областей оперативної пам'яті, що виділені операційною системою процесу, називається його ***адресним простором***. Говорять, що кожен процес працює у своєму адресному просторі, маючи на увазі захист адресних просторів, який здійснюється ОС. Захищаються і інші типи ресурсів, такі як файли, зовнішні пристрої тощо. ОС може не лише захищати ресурси, що виділені одному процесу, але і організовувати їх спільне використання, наприклад, дозволяти доступ до деякої області пам'яті декільком процесам.

Упродовж періоду існування процесу його виконання може бути багаторазово перерване і продовжене. Для того щоб відновити виконання процесу, необхідно відновити стан його операційного середовища. Стан операційного середовища ідентифікується станом регістрів і програмного лічильника, режимом роботи процесора, покажчиками на відкриті файли, інформацією про незавершені операції введення-виведення, кодами помилок виконуваних цим процесом системних викликів тощо. Ця інформація називається ***контекстом процесу.*** Говорять, що при зміні процесу відбувається перемикання контекстів.

Операційна система бере на себе також функції синхронізації процесів, що дозволяють процесу призупиняти своє виконання до настання якої-небудь події в системі, наприклад, завершення операції введення-виведення, здійснюється за її запитом операційній системою.

Для реалізації складних програмних комплексів корисно буває організувати їх роботу у вигляді декількох паралельних процесів, які періодично взаємодіють один з одним і обмінюються деякими даними. Оскільки ОС захищає ресурси процесів і не дозволяє одному процесу писати в пам'ять або читати з пам'яті іншого процесу, то для оперативної взаємодії процесів ОС повинна надавати особливі засоби, які називають засобами міжпроцесної взаємодії.

Таким чином, підсистема управління процесами планує виконання процесів, тобто розподіляє процесорний час між декількома одночасно існуючими в системі процесами, займається створенням і знищенням процесів, забезпечує процеси необхідними системними ресурсами, підтримує синхронізацію процесів, а також забезпечує взаємодію між процесами.

2.2.5 Управління пам'яттю

Пам'ять для процесу є таким же важливим ресурсом, як і процесор, оскільки процес може виконуватися процесором тільки в тому випадку, якщо його коди і дані (не обов'язково всі) знаходяться в оперативній пам'яті.

Управління пам'яттю включає розподіл наявної фізичної пам'яті між усіма існуючими в системі в даний момент процесами, завантаження кодів і цих процесів у відведені їм області пам'яті, налаштування адресно-залежних частин кодів процесу на фізичні адреси виділеної області, а також захист областей пам'яті кожного процесу.

Існує велика різноманітність алгоритмів розподілу пам'яті. Вони можуть відрізнятися, наприклад, кількістю областей пам'яті, що виділяються процесу. В одних випадках пам'ять виділяється процесу у вигляді однієї безперервної області, а в інших – у вигляді декількох несуміжних областей. У деяких системах розподіл пам'яті здійснюється сторінками фіксованого розміру, а в інших – сегментами змінної довжини.

Одним з найпопулярніших способів управління пам'яттю в сучасних операційних системах є так звана віртуальна пам'ять. Наявність в ОС механізму віртуальної пам'яті дозволяє програмістові писати програму так, ніби в його розпорядженні є однорідна оперативна пам'ять великого об'єму, що часто істотно перевищує об'єм наявної фізичної пам'яті. Насправді усі дані, що використовуються програмою, зберігаються на диску і при необхідності частинами (сегментами або сторінками) відображаються у фізичну пам'ять.

При переміщенні кодів і даних між оперативною пам'яттю і диском підсистема віртуальної пам'яті виконує трансляцію віртуальних адрес, отриманих в результаті компіляції і компонування програми, у фізичні адреси елементів оперативної пам'яті. Захист пам'яті – це вибіркова здатність оберігати виконуване завдання від запису або читання пам'яті, призначеної іншому завданню. Правильно написані програми не намагаються звертатися до пам'яті, призначеної іншим процесам. Проте реальні програми часто містять помилки, в результаті яких такі спроби іноді робляться. Засоби захисту пам'яті, які реалізовані в операційній системі, повинні забороняти несанкціонований доступ процесів до чужих областей пам'яті.

Таким чином, функціями ОС з управління пам'яттю є: відстежування вільної і зайнятої пам'яті; виділення пам'яті процесам і звільнення пам'яті при завершенні процесів; захист пам'яті; витіснення процесів з оперативної пам'яті на диск, а також налаштування адрес програми на конкретну область фізичної пам'яті.

2.2.6 Управління файлами і зовнішніми пристроями

Здатність ОС до «екранування» складнощів реальної апаратури дуже яскраво проявляється в одній з основних підсистем ОС – файловій системі. Операційна система віртуалізує окремий набір даних, що зберігаються на зовнішньому накопичувачі, у вигляді файлу – простої неструктурованої послідовності байтів, що має символьне ім'я. Для зручності роботи з даними файли групуються в каталоги, які, у свою чергу, утворюють групи – каталоги вищого рівня. Користувач може за допомогою ОС виконувати над файлами і каталогами такі дії, як пошук за іменем, видалення, виведення вмісту на зовнішній пристрій (наприклад, на дисплей), зміну і збереження вмісту.

Файлова система ОС виконує перетворення символьних імен файлів, з якими працює користувач або прикладний програміст, у фізичні адреси даних на диску, організовує спільний доступ до файлів, захищає їх від несанкціонованого доступу.

При виконанні своїх функцій файлова система тісно взаємодіє з підсистемою управління зовнішніми пристроями, яка за запитами файлової системи здійснює передачу даних між дисками і оперативною пам'яттю.

Підсистема управління зовнішніми пристроями, що називається також підсистемою введення-виведення, виконує роль інтерфейсу до усіх пристроїв, підключених до комп'ютера. Спектр цих пристроїв дуже великий. Номенклатура накопичувачів, що випускаються, на жорстких, гнучких і оптичних дисках, принтерів, сканерів, моніторів, плотерів, модемів, мережевих адаптерів і більше спеціальних облаштувань введення-виведення, таких як, наприклад, аналого-цифрові перетворювачі, може налічувати сотні моделей. Ці моделі можуть істотно відрізнятися набором і послідовністю команд, за допомогою яких здійснюється обмін інформацією з процесором і пам'яттю комп'ютера, швидкістю роботи, кодуванням передаваних даних, можливістю спільного використання і певною кількістю інших деталей.

Програма, що управляє конкретною моделлю зовнішнього пристрою і враховує усі його особливості, називається ***драйвером*** цього пристрою (від англійського drive – управляти, вести). Драйвер може управляти єдиною моделлю пристрою, наприклад модемом U-1496E компаній ZyXEL, або ж групою пристроїв певного типу. Для користувача дуже важливо, щоб операційна система включала якомога більше різноманітних драйверів, оскільки це гарантує можливість підключення до комп'ютера великого числа зовнішніх пристроїв різних виробників. Від наявності відповідних драйверів багато в чому залежить успіх операційної системи на ринку. Наприклад, відсутність багатьох необхідних драйверів зовнішніх пристроїв була однією з причин низької популярності OS/2.

Створенням драйверів пристроїв займаються як розробники конкретної ОС, так і фахівці компаній, що випускають зовнішні пристрої. Операційна система повинна добре підтримувати певний інтерфейс між драйверами і іншою частиною ОС, щоб розробники з компаній-виробників пристроїв введення-виведення могли поставляти разом зі своїми пристроями драйвери для цієї операційної системи.

Прикладні програмісти можуть користуватися інтерфейсом драйверів при розробці своїх програм. Але це не дуже зручно – такий інтерфейс є низькорівневими операціями, обтяженими великою кількістю деталей.

Підтримка високорівневого уніфікованого інтерфейсу прикладного програмування до різнорідних пристроїв введення-виведення є одним з найважливіших завдань ОС. З часу появи ОС UNIX такий уніфікований інтерфейс у більшості ОС будується на основі концепції файлового доступу. Ця концепція полягає в тому, що обмін з будь-яким зовнішнім пристроєм виглядає як обмін з файлом, що має ім'я і є неструктурованою послідовністю байтів. Файлом може виступати як реальний файл на диску, так і алфавітно-цифровий термінал, друкуючий пристрій або мережевий адаптер.

2.2.7 Захист даних і адміністрування

Безпека даних обчислювальної системи забезпечується засобами відмовостійкості ОС, спрямованими на захист від збоїв і відмов апаратури і помилок програмного забезпечення, а також засобами захисту від несанкціонованого доступу.

Першою при захисті даних від несанкціонованого доступу є процедура логічного входу. ОС повинна переконатися, що до системи намагається увійти користувач, вхід якого дозволений адміністратором. Функції захисту ОС взагалі дуже тісно пов'язані з функціями адміністрування, оскільки саме адміністратор визначає права користувачів при їх зверненні до різних ресурсів системи – файлів, каталогів, принтерів, сканерам тощо. Крім того, адміністратор обмежує можливості користувачів у виконанні тих або інших системних дій.

Наприклад, користувачеві може бути заборонено виконувати процедуру завершення роботи ОС, встановлювати системний час, завершувати чужі процеси, створювати облікові записи користувачів, змінювати права доступу до деяких каталогів і файлів. Адміністратор може також урізувати можливості інтерфейсу користувача, прибравши, наприклад, деякі пункти з меню операційної системи, що виводиться на дисплей користувача.

Важливим засобом захисту даних є ***функції аудиту ОС***, що полягають у фіксації усіх подій, від яких залежить безпека системи. Наприклад, спроби вдалого і невдалого логічного входу в систему, операції доступу до деяких каталогів і файлів, використання принтерів тощо. Список подій, які необхідно відстежувати, визначає адміністратор ОС.

Підтримка відмовостійкості реалізується ОС, як правило, на основі резервування. Найчастіше у функції ОС входить підтримка декількох копій даних на різних дисках або різних дискових накопичувачах. Резервуються також принтери і інші пристрої введення-виведення. При відмові одного з надлишкових пристроїв ОС повинна швидко і прозорим для користувача способом зробити реконфігурацію системи і продовжити роботу з резервним пристроєм. Особливим випадком забезпечення відмовостійкості є використання декількох процесорів, тобто мультипроцесування, коли система продовжує роботу при відмові одного з процесорів. Підтримка відмовостійкості входить в

обов'язки системного адміністратора. До складу ОС входять утиліти, що дозволяють адміністраторові виконувати регулярні операції резервного копіювання для забезпечення швидкого відновлення важливих даних.

2.2.8 Інтерфейс прикладного програмування

Прикладні програмісти використовують у своїх додатках звернення до ОС, коли для виконання тих або інших дій їм потрібен особливий статус, який має тільки операційна система. Наприклад, у більшості сучасних ОС усі дії, які пов'язані з управлінням апаратними засобами комп'ютера, може виконувати тільки ОС. Окрім цих функцій прикладний програміст може скористатися набором сервісних функцій ОС, які спрощують написання додатків. Функції такого типу реалізують універсальні дії, що часто вимагаються в різних додатках, такі, наприклад, як обробка текстових рядків. Ці функції могли б бути виконані і самим додатком, проте набагато простіше використати вже готові, відлагоджені процедури, включені до складу операційної системи. В той же час навіть за наявності в ОС відповідної функції програміст може реалізувати її самостійно в рамках додатка, якщо запропонований операційною системою варіант його не цілком влаштовує.

Можливості операційної системи, що доступні прикладному програмістові у вигляді набору функцій, називаються ***інтерфейсом прикладного програмування*** (Application Programming Interface, **API**). Від кінцевого користувача ці функції приховані за оболонкою алфавітно-цифрового або графічного інтерфейсу користувача.

Для розробників додатків усі особливості конкретної операційної системи представлені особливостями її API. Тому операційні системи з різною внутрішньою організацією, але з однаковим набором функцій API, здаються нам однією і тією ж ОС, що спрощує стандартизацію операційних систем і забезпечує переносимість додатків між різними ОС, такими, що відповідають певному стандарту на API.

Наприклад, наслідування загальних стандартів API UNIX, одним з яких є стандарт **Posix** (Portable Operating System Interface – Інтерфейс переносимих операційних систем), дозволяє говорити про деяку узагальнену операційну систему UNIX, хоча численні версії цієї ОС від різних виробників іноді істотно відрізняються внутрішньою організацією.

Додатки виконують звернення до функцій API за допомогою системних викликів. Спосіб, яким додаток отримує послуги операційної системи, дуже схожий на виклик підпрограм. Інформація, яка потрібна ОС і яка складається з ідентифікатора команди і даних, поміщається в певне місце пам'яті, в регістри і/або стек. Потім управління передається операційній системі, яка виконує необхідну функцію і повертає результати через пам'ять, регістри або стеки. Якщо операція проведена не успішно, то результат включає індикацію помилки.

2.2.9 Інтерфейс користувача

Операційна система повинна забезпечувати зручний інтерфейс не лише для застосовних програм, але і для людини, працюючої за терміналом. Ця людина може бути кінцевим користувачем, адміністратором ОС або програмістом.

У ранніх операційних системах пакетного режиму функції інтерфейсу користувача були зведені до мінімуму і не вимагали наявності терміналу. Команди мови управління завданнями набивалися на перфокарти, а результати виводилися на друкуючий пристрій.

Сучасні ОС підтримують розвинені функції інтерфейсу користувача для інтерактивної роботи за терміналами двох типів: алфавітно-цифровими і графічними.

При роботі за алфавітно-цифровим терміналом користувач має у своєму розпорядженні систему команд, потужність яких відображає функціональні можливості цієї ОС. Командна мова ОС дозволяє запускати і зупиняти додатки, виконувати різні операції з файлами і каталогами, отримувати інформацію про стан ОС (кількість працюючих процесів, об'єм вільного простору на дисках тощо), адмініструвати систему. Команди можуть вводитися не лише в інтерактивному режимі з терміналу, але і прочитуватися з так званого командного файлу, що містить деяку послідовність команд.

Програмний модуль ОС, відповідальний за читання окремих команд або послідовності команд з командного файлу, іноді називають командним інтерпретатором.

Введення команди може бути спрощене, якщо операційна система підтримує графічний інтерфейс користувача. В цьому випадку користувач для виконання потрібної дії за допомогою миші вибирає на екрані потрібний пункт меню або графічний символ.

2.3 Можливості розвитку ОС

Більшість ОС постійно розвиваються. Для цього є цілий ряд причин.

1. **Оновлення і виникнення нових видів апаратного забезпечення**. Наприклад, ранні версії ОС UNIX не використали механізму сторінкової організації пам'яті, оскільки вони функціонували на комп'ютерах, які не були забезпечені відповідною апаратною підтримкою. Наступне покоління ОС були допрацьовані так, щоб вони могли використати нові апаратні можливості.
2. **Нові сервіси і можливості**. Прикладом може бути підтримка нових додатків, які використовують вікна на екрані дисплея. Ця можливість спричинила за собою значне оновлення ОС.
3. **Виправлення**. У кожній ОС є помилки. Час від часу вони виявляються і виправляються. Необхідність внесення регулярних змін в ОС накладає певні вимоги до їх структури. Ці системи повинні мати модульну структуру з певною взаємодією модулів.

2.4 Компоненти комп'ютерної системи

Щоб краще зрозуміти місце і роль операційної системи в процесі обчислень, розглянемо комп'ютерну систему в цілому. Вона складається з таких компонент.

1. **Апаратура** (hardware) комп'ютера, до основних частин якої належать:
   - центральний процесор (Central Processor Unit – CPU), що виконує команди (інструкції) комп'ютера;
   - пам'ять (memory), що зберігає дані і програми;
   - пристрої введення-виведення, або зовнішні пристрої (input-output devices, I/O devices), що забезпечують введення інформації в комп'ютер і виведення результатів роботи програм у формі, що сприймається користувачем-людиною або іншими програмами.

2. Операційна система – системне програмне забезпечення, що управляє використанням апаратури комп'ютера, різними програмами і користувачами.

3. Прикладне програмне забезпечення (applications software) – програми, призначені для розв'язання різних класів задач. До них належать, зокрема:

- компілятори, що забезпечують трансляцію програм з мов програмування в машинний код;
- системи управління базами даних (СУБД);
- графічні бібліотеки, ігрові програми, офісні програми.

Прикладне програмне забезпечення утворює наступний, вищий рівень, в порівнянні з операційною системою, і дозволяє розв'язувати на комп'ютері різні прикладні і повсякденні задачі.

4. Користувачі (users) – люди та інші комп'ютери. Віднесення користувача-людини до компонент комп'ютерної системи – це реальність, оскільки будь-який користувач фактично стає частиною обчислювальної системи в процесі своєї роботи на комп'ютері.

Одна з важливих функцій ОС якраз і полягає в тому, щоб позбавити користувача від більшої частини такої рутинної роботи (наприклад, резервного копіювання файлів) і дозволити йому зосередитися на творчій роботі.

2.5 Класифікація ОС

Операційні системи можуть відрізнятися особливостями реалізації внутрішніх алгоритмів управління основними ресурсами комп'ютера (процесорами, пам'яттю, пристроями), особливостями використаних методів проектування, типами апаратних платформ, областями використання і багатьма іншими властивостями. Нижче приведена класифікація ОС за декількома основними ознаками.

2.5.1 Особливості алгоритмів управління ресурсами

Від ефективності алгоритмів управління локальними ресурсами комп'ютера багато в чому залежить ефективність усієї ОС в цілому. Тому, характеризуючи ОС, часто наводять найважливіші особливості реалізації

функцій ОС з управління процесорами, пам'яттю, зовнішніми прстроями автономного комп'ютера. Так, наприклад, залежно від особливостей використаного алгоритму управління процесором, операційні системи ділять на ***багатозадачні,*** розраховані на багато користувачів і ***однозадачні,*** розраховані на одного користувача, на системи, що підтримують ***багатопотокову*** обробку і не підтримують її, на ***багатопроцесорні*** і ***однопроцесорні*** системи.

Підтримка багатозадачності. За числом одночасно виконуваних завдань операційні системи можуть бути розділені на два класи:

1. Однозадачні (наприклад, MS-DOS).
2. Багатозадачні (ОС ЕС, OS/2, UNIX, Windows 9x).

Однозадачні ОС, в основному, виконують функцію надання користувачеві віртуальної машини, роблячи простішим і зручнішим процес взаємодії користувача з комп'ютером.

Багатозадачні ОС, окрім вищеперелічених функцій, управляють розподілом спільно використовуваних ресурсів, таких як процесор, оперативна пам'ять, файли і зовнішні пристрої.

Підтримка режиму, розрахованого на багато користувачів. По числу одночасно працюючих користувачів ОС діляться на:

1. Розраховані на одного користувача (MS-DOS, Windows 3.x, ранні версії OS/2).
2. Розраховані на багато користувачів (UNIX, Windows NT).

Головною відмінністю систем, розрахованих на багато користувачів, від систем, розрахованих на одного користувача, є наявність засобів захисту інформації кожного користувача від несанкціонованого доступу інших користувачів. Слід зауважити, що не всяка багатозадачна ОС може бути розрахованою на багато користувачів, і не всяка ОС, розрахована на одного користувача, є однозадачною.

Витісняюча і невитісняюча багатозадачність. Найважливішим розподіленим ресурсом є процесорний час. Спосіб розподілу процесорного часу між декількома одночасно існуючими в системі процесами (чи потоками) багато в чому визначає специфіку ОС. Серед множини існуючих варіантів реалізації багатозадачності можна виділити дві групи алгоритмів:

1. Невитісняюча багатозадачність (NetWare, Windows 3.x).
2. Витісняюча багатозадачність (Windows NT, OS/2, UNIX).

Основною відмінністю між витісняючими і не витісняючими варіантами багатозадачності є міра централізації механізму планування процесів. У першому випадку механізм планування процесів цілком зосереджений в операційній системі, а в другому – розподілений між системою і прикладними програмами.

При невитісняючій багатозадачності активний процес виконується до тих пір, поки він сам, за власною ініціативою, не віддасть управління операційній системі для того, щоб та вибрала з черги інший готовий до виконання процес.

При витісняючій багатозадачності рішення про перемикання процесора з одного процесу на інший приймається операційною системою, а не самим активним процесом.

Підтримка багатопоточності. Важливою властивістю операційних систем є можливість розпаралелювання обчислень у рамках одного завдання. Багатопотокова ОС розподіляє процесорний час не між задачами, а між їх окремими гілками (потоками, нитками).

Багатопроцесорна обробка. Іншою важливою властивістю ОС є відсутність або наявність в ній засобів підтримки багатопроцесорної обробки – ***мультипроцесування***. Мультипроцесування призводить до ускладнення всіх алгоритмів управління ресурсами. Такі функції є в операційних системах Solaris 2.x фірми Sun, OS/2 фірми IBM, Windows NT фірми Microsoft.

2.5.2 Особливості апаратних платформ

На властивості операційної системи безпосередній вплив роблять апаратні засоби, на які вона орієнтована. За типом апаратури відрізняють операційні системи персональних комп'ютерів, міні-комп'ютерів, мейнфреймів, кластерів.

Кластер – слабко зв'язана сукупність декількох обчислювальних систем, що представляються користувачеві єдиною системою, і працюють спільно для виконання загальних додатків. Серед перерахованих типів комп'ютерів можуть зустрічатися як однопроцесорні варіанти, так і багатопроцесорні. У будь-якому випадку специфіка апаратних засобів, як правило, відображається на специфіці операційних систем.

2.5.3 Особливості областей використання

Багатозадачні ОС підрозділяються на три типи відповідно до використаних при їх розробці критеріїв ефективності:

1. Системи пакетної обробки (наприклад, ОС ЕС, IBM/360).
2. Системи розподілу часу (UNIX, WinXP).
3. Системи реального часу (QNX фірми Quantum Software systems, RT-11 для міні-комп'ютерів PDP-11).

Системи пакетної обробки призначалися для розв'язання задач в основному обчислювального характеру, що не вимагають швидкого отримання результатів. Головною метою і критерієм ефективності систем пакетної обробки є максимальна пропускна спроможність, тобто розв'язок максимального числа завдань в одиницю часу (ОС ЄС, IBM/360).

Системи розподілу часу покликані виправити основний недолік систем пакетної обробки – ізоляцію користувача-програміста від процесу виконання його завдань. Кожному користувачеві системи розподілу часу надається термінал, з якого він може вести діалог зі своєю програмою. Оскільки в системах розподілу часу кожному завданню виділяється тільки ***квант*** процесорного часу, жодне завдання не займає процесор надовго, і час відповіді виявляється прийнятним.

Якщо квант вибраний досить невеликим, то в усіх користувачів, одночасно працюючих на одній і тій же машині, складається враження, що кожен з них одноосібно використовує машину. Ясно, що системи розподілу часу мають меншу пропускну спроможність, чим системи пакетної обробки, оскільки на

виконання приймається кожне запущене користувачем завдання, а не те, яке «вигідне» системі, і, крім того, є накладні витрати обчислювальної потужності на частіше перемикання процесора із задачі на задачу. Критерієм ефективності систем розподілу часу є не максимальна пропускна спроможність, а зручність і ефективність роботи користувача (UNIX, WinXP).

Системи реального часу застосовуються для управління різними технічними об'єктами, такими, наприклад, як верстат, супутники тощо. В усіх цих випадках існує гранично допустимий час, впродовж якого має бути виконана та або інша програма, що управляє об'єктом. Інакше може статися аварія: супутник вийде із зони видимості, експериментальні дані, що поступають з датчиків, будуть втрачені. Таким чином, критерієм ефективності для систем реального часу є їх здатність витримувати заздалегідь задані інтервали часу між запуском програми і отриманням результату. Цей час називається ***часом реакції системи,*** а відповідна властивість системи – ***реактивністю.*** Для цих систем мультипрограмна суміш є фіксованим набором заздалегідь розроблених програм, а вибір програми на виконання здійснюється виходячи з поточного стану об'єкту.

Вбудовані ОС. До них належать управляючі програми для різних мікропроцесорних систем, які використовуються у військовій техніці, в побутовій електроніці, смарт-картах тощо. Вбудовані ОС розробляються під конкретний пристрій. До таких систем належить Embedded Linux і Windows CE.

2.5.4 Особливості методів побудови

При опису операційної системи часто вказуються особливості її структурної організації і основні концепції, що покладені в її основу. До таких базових концепцій належать перелічені нижче.

Способи побудови ядра системи – монолітне ядро чи мікроядро. Більшість ОС використовують ***монолітне ядро***, яке компонується як одна програма, працююча в привілейованому режимі, і яка використовує швидкі переходи з однієї процедури на іншу, не вимагаючи перемикання з привілейованого режиму в режим користувача і навпаки.

Альтернативою є побудова ОС на базі ***мікроядра.*** Мікроядро працює в привілейованому режимі і виконує тільки мінімум функцій з управління апаратурою. Функції ОС вищого рівня виконують спеціалізовані компоненти ОС – ***сервери***, працюючі в режимі користувача. При такій побудові ОС працює повільніше, оскільки часто виконуються переходи між привілейованим режимом і режимом користувача. Зате система виходить гнучкішою – її функції можна нарощувати, модифікувати або звужувати, додаючи, модифікуючи або виключаючи сервери режима користувача. Крім того, сервери добре захищені один від одного, як і будь-які процеси користувача.

Побудова ОС на базі об'єктно-орієнтованого підходу дає можливість використати всі його переваги, що добре зарекомендували себе на рівні додатків, усередині операційної системи, до яких можна віднести:

- акумуляцію вдалих рішень у формі стандартних об'єктів;

- можливість створення нових об'єктів на базі наявних за допомогою механізму ***спадкоємства***;
- хороший захист даних за рахунок їх ***інкапсуляції*** у внутрішні структури об'єкту, що робить дані недоступними для несанкціонованого використання ззовні.

Наявність декількох операційних (прикладних) середовищ дає можливість у рамках однієї ОС одночасно виконувати додатки, розроблені для декількох ОС. Багато сучасних ОС підтримують одночасно прикладні середовища MS-DOS, Windows, UNIX, OS/2 або хоч би деякої підмножини з цього популярного набору. Концепція множинних прикладних середовищ найпростіше реалізується в ОС на базі мікроядра, над яким працюють різні сервери, частину яких реалізує прикладне середовище тієї або іншої ОС.

Розподілена організація ОС дозволяє спростити роботу користувачів і програмістів в мережевих середовищах. У розподіленій ОС реалізовані механізми, які дають можливість користувачеві представляти мережу у вигляді традиційного однопроцесорного комп'ютера. Ознаками розподіленої організації ОС є: наявність єдиної довідкової служби розподілу ресурсів, єдиної служби часу, багатопотокової обробки, що дозволяє розпаралелювати обчислення в рамках одного завдання і виконувати це завдання відразу на декількох комп'ютерах мережі.

Контрольні питання і тести до розділу 2

Контрольні питання

1. Який комплекс управляючих і обробляючих програмних засобів розуміють під операційною системою?
2. Дайте визначення операційного середовища.
3. Назвіть основні функції операційних систем.
4. Дайте визначення ОС як віртуальної машини.
5. Назвіть основні функції ОС як диспетчера ресурсів.
6. Які ресурси повинна призначити ОС процесу, щоб він міг бути виконаний?
7. Сукупність яких ресурсів називають адресним простором процесу?
8. Яка інформація належить до контексту процесу?
9. Перчисліть основні функції підсистеми управління пам'яттю.
10. Перечисліть основні функції підсистеми управління файлами і зовнішніми пристроями.
11. Дайте визначення інтерфейсу прикладного програмування (API).
12. Що таке стандарт Posix?
13. За якими основними ознаками класифікують ОС?
14. Що таке витісняюча і невитісняюча багатозадачність?
15. Назвіть особливості методів побудови ОС.
16. Операційні системи управляють тільки апаратними засобами? (Так, Ні)
17. Чи вірно, що основним завданням операційних систем є організація зручного інтерфейсу користувача? (Так, Ні)

Тести

1. Одна операційна система може підтримувати декілька:
   1) мікропрограмних середовищ;
   2) операційних систем;
   3) операційних середовищ;
   4) мікропрограмних систем.
2. Інтерфейс прикладного програмування призначений для використання прикладними програмами:
   1) системних ресурсів комп'ютера;
   2) інтерпретатора команд користувача;
   3) регістрів загального призначення процесора;
   4) адресного простору процесу.
3. Операційна система належить до:
   1) прикладного програмного забезпечення;
   2) системного програмного забезпечення;
   3) інструментального програмного забезпечення.
4. Операційною системою називається:
   1) програма управління операціями користувача;
   2) програмне середовище з графічним інтерфейсом;
   3) прикладна програма, призначена для виконання специфічних функцій;
   4) система, що надає користувачеві інтерфейс до ПК і управляє його ресурсами.
5. Цілі операційної системи:
   1) забезпечити зручність, ефективність, надійність і безпеку використання комп'ютерного устаткування, зовнішніх пристроїв і призначених для користувача програм;
   2) забезпечити зберігання і резервне копіювання даних користувача;
   3) забезпечити ефективне і швидке досягнення користувачем своїх професійних цілей з використанням зручних і надійних апаратних і програмних інструментів.
6. Комплекс системних управляючих і оброблювальних програм, призначених для ефективного використання усіх ресурсів обчислювальної системи і зручності роботи з нею, називається:
   1) операційним середовищем;
   2) управлячим середовищем;
   3) операційною системою.
7. У мікроядерній ОС функції ОС вищого рівня виконують спеціалізовані компоненти ОС:
   1) утиліти;
   2) сервери;
   3) мікроядро.

3 СУЧАСНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ОС

Операційна система є серцевиною програмного забезпечення, вона створює середовище для виконання додатків і багато в чому визначає, які корисні для користувача властивості матимуть ці додатки. У зв'язку з цим розглянемо вимоги, яким повинна задовольняти сучасна ОС.

3.1 Вимоги, що пред'являються до операційної системи

Очевидно, що головною вимогою, що пред'являється до операційної системи, є здатність виконання основних функцій: ефективного управління ресурсами і забезпечення зручного інтерфейсу для користувача і прикладних програм. Сучасна ОС, як правило, повинна реалізовувати мультипрограмну обробку, віртуальну пам'ять, свопінг, підтримувати багатовіконний інтерфейс, а також виконувати багато інших, абсолютно необхідних функцій.

Окрім цих вимог функціональної повноти до операційних систем пред'являються не менш важливі експлуатаційні і ринкові вимоги, які наведені нижче.

Розширюваність. Код ОС має бути написаний так, щоб можна було легко внести доповнення і зміни, якщо це знадобиться, і не порушити цілісність системи.

Переносимість. Код ОС повинен легко переноситися з процесора одного типу на процесор іншого типу і з апаратної платформи одного типу на апаратну платформу іншого типу.

Надійність і відмовостійкість. Система має бути захищена як від внутрішніх, так і від зовнішніх помилок, збоїв і відмов. Її дії мають бути завжди передбачуваними, а додатки не зможуть завдавати шкоди ОС. Надійність і відмовостійкість ОС передусім визначаються архітектурними рішеннями, покладеними в її основу, а також якістю її реалізації. Крім того, важливо, чи включає ОС програмну підтримку апаратних засобів забезпечення відмовостійкості, таких, наприклад, як дискові масиви або джерела безперебійного живлення.

Сумісність. ОС повинна мати засоби для виконання прикладних програм, написаних для інших операційних систем. Крім того, призначений для користувача інтерфейс має бути сумісний з існуючими системами і стандартами.

Захист інформації і безпека. ОС повинна мати засоби захисту ресурсів одних користувачів від інших.

Продуктивність. Система повинна мати настільки хорошу швидкодію і час реакції, наскільки це дозволяє апаратна платформа. На продуктивність ОС впливає багато чинників, серед яких основними є архітектура ОС, різноманіття функцій, якість програмування коду, можливість виконання ОС на високопродуктивній (багатопроцесорній) платформі тощо.

Постійне зростання вимог до ОС призводить не лише до удосконалення їх архітектури, але і до нових способів їх організації. Як вже відзначалося вище, для задоволення вимог, що пред'являються до сучасної ОС, велике значення має її структурна побудова. Операційні системи пройшли тривалий шлях розвитку від

монолітних систем до добре структурованих модульних систем, здатних до розвитку, розширення і легкого перенесення на нові платформи. В експериментальних і комерційних ОС були випробувані найрізноманітніші підходи і структурні елементи, більшість з яких можна об'єднати в наступні категорії:

1. Архітектура ядра.
2. Множинні прикладні середовища.
3. Концепція віртуальних машин.
4. Мережеві і розподілені ОС.
5. Симетрична багатопроцесорність.
6. Багатопоточність.

Розглянемо детальніше деякі з цих вимог і структурну побудову сучасних операційних систем.

3.2 Розширюваність

Тоді як апаратна частина комп'ютера застаріває за декілька років, корисне життя операційних систем може вимірюватися десятиліттями. Тому операційні системи завжди еволюційно змінюються з часом, і ці зміни значиміші, ніж зміни апаратних засобів. Зміни ОС є набуттям нею нових властивостей. Наприклад, підтримка нових пристроїв, таких як CD-ROM, можливість зв'язку з мережами нового типу, підтримка багатообіцяючих технологій, таких як графічний інтерфейс користувача або об'єктно-орієнтоване програмне оточення, використання більш ніж одного процесора. Тому збереження цілісності коду, які б зміни не вносилися в операційну систему, є головною метою розробки.

Розширюваність може досягатися за рахунок модульної структури ОС, при якій програми будуються з набору окремих модулів, що взаємодіють тільки через функціональний інтерфейс. Нові компоненти можуть бути додані в операційну систему модульним шляхом, вони виконують свою роботу, використовуючи інтерфейси, підтримувані існуючими компонентами.

Використання об'єктів для представлення системних ресурсів також покращує розширюваність системи. ***Об'єкти*** – це абстрактні типи даних, над якими можна виконувати тільки ті дії, які передбачені спеціальним набором об'єктних функцій. Об'єкти дозволяють однаково управляти системними ресурсами. Додавання нових об'єктів не руйнує існуючі об'єкти і не вимагає змін існуючого коду.

Прекрасні можливості для розширення надає підхід до структуризації ОС за типом клієнт-сервер з використанням технології мікроядра. Відповідно до цього підходу ОС будується як сукупність привілейованої програми, що управляє, і набору непривілейованих послуг-серверів. Основна частина ОС може залишатися незмінною в той час, як можуть бути додані нові сервери або удосконалені старі.

3.3 Переносимість

В ідеалі код ОС повинен легко переноситися з процесора одного типу на процесор іншого типу, і з апаратної платформи одного типу на апаратну платформу іншого типу. Переносимі ОС мають декілька варіантів реалізації для різних платформ, таку властивість ОС називають також багатоплатформністю (кросплатформністю, мультиплатформністю).

Вимога переносимості коду тісно пов'язана також з розширюваністю. Розширюваність дозволяє покращувати операційну систему, тоді як переносимість дає можливість переміщати усю систему на машину, що базується на іншому процесорі або апаратній платформі, роблячи при цьому по можливості невеликі зміни в коді. Написання переносимої ОС аналогічно написанню будь-якого переносимого коду. Для цього треба дотримуватись деяких правил.

По-перше, велика частина коду має бути написана мовою, яка є на усіх машинах, куди треба переносити систему. Це означає, що код має бути написаний на мові високого рівня, переважно стандартизованою, наприклад, на мові *С*. Програма, написана на асемблері, не є переносимою, якщо тільки переносити її на машину, що має командну сумісність з машиною, з якої переносять додаток.

По-друге, слід врахувати, в яке фізичне оточення програма має бути перенесена. Різна апаратура вимагає різних рішень при створенні ОС. Наприклад, ОС, побудована на 32-бітових адресах, не може бути перенесена на машину з 16-бітовими адресами (хіба що з величезними труднощами).

По-третє, важливо мінімізувати або, якщо можливо, виключити ті частини коду, які безпосередньо взаємодіють з апаратними засобами.

По-четверте, якщо код, залежний від апаратури, не може бути повністю виключений, то він має бути ізольований в декількох модулях, що добре локалізуються.

Для легкого перенесення ОС при її розробці мають бути дотримані такі вимоги:

1. **Переносима мова високого рівня.** Більшість переносимих ОС написані на мові **С** (стандарт ANSI X3.159-1989). Розробники вибирають мову **С** тому, що вона стандартизована, і тому, що С-компілятори широко доступні. Асемблер використовується тільки для тих частин системи, які повинні безпосередньо взаємодіяти з апаратурою (наприклад, обробник переривань) або для частин, які вимагають максимальної швидкості (наприклад, цілочисельна арифметика підвищеної точності). Проте непереносний код має бути ретельно ізольований усередині тих компонентів, де він використовується.
2. **Ізоляція процесора**. Деякі низькорівневі частини ОС повинні мати доступ до структур даних і регістрів, залежних від процесора. Проте код, який робить це, повинен міститися в невеликих модулях, які можуть бути замінені аналогічними модулями для інших процесорів.

3.4 Сумісність

Існує декілька «довгоживучих» популярних операційних систем (різновиди UNIX, MS-DOS, Windows, OS/2), для яких напрацьована широка номенклатура додатків. Деякі з них користуються широкою популярністю. Тому для користувача, що переходить з однієї ОС або на іншу апаратну платформу, дуже приваблива можливість запуску в новому середовищі звичних додатків. Якщо ОС має засоби для виконання прикладних програм, написаних для ОС, то про неї говорять, що вона має ***сумісність*** з цими ОС. Поняття сумісності включає також підтримку інтерфейсів користувача інших ОС.

Слід розділяти питання ***двійкової сумісності*** і сумісності на ***рівні початкових текстів додатків***. Двійкова сумісність досягається в тому випадку, коли можна взяти виконувану програму і запустити її на виконання на іншій ОС. Для цього потрібні: сумісність на рівні команд процесора, сумісність на рівні системних викликів і навіть на рівні бібліотечних викликів, якщо вони є динамічно зв'язуваними.

Сумісність на рівні початкових текстів вимагає наявності відповідного компілятора в складі програмного забезпечення, а також сумісності на рівні бібліотек і системних викликів. При цьому потрібна перекомпіляція наявних початкових текстів в новий виконуваний модуль.

Сьогодні таку сумісність забезпечує стандартизація розробки ПО:

- наявність стандарту на мови програмування;
- наявність стандарту на інтерфейс операційних систем.

Роботи щодо стандартизації інтерфейсу ОС відбуваються в рамках проекту POSIX (Portable Operating System Interfice – Переносимий Інтерфейс ОС). Найважливішим стандартом є POSIX 1003.1, який описує набір бібліотечних процедур (відкриття файлу, створення нового процесу і тому подібне), які можуть бути реалізовані в системі. Цей процес стандартизації триває і в наші дні. Останньою модифікацією стандарту є базова специфікація Open Group/IEEE. Ці стандарти відображають традиційний набір засобів, реалізований в UNIX-сумісних системах.

Так, функції бібліотеки підсистеми Win32 API виконуються в режимі користувача і в режимі ядра. Наприклад, до виходу Windows NT 4.0 (1996 р.) віконна і графічна системи працювали в режимі користувача як частину процесу підсистеми Win32. Потім для підвищення продуктивності реалізацію цих підсистем було перенесено в режим ядра.

Підсистема POSIX працює в режимі користувача і реалізує набір функцій, визначених стандартом POSIX 1003.1. Оскільки додатки, написані для однієї підсистеми, не можуть використати функції інших підсистем, в POSIX-програмах не можна користуватися засобами Win32 API (зокрема, графічними і мережевими функціями), що знижує значущість цієї підсистеми. Підсистема POSIX не є обов'язковим компонентом Windows XP.

Чи має нова ОС двійкову сумісність або сумісність початкових текстів з існуючими системами, залежить від багатьох чинників. Найголовніший з них – ***архітектура процесора***, на якому працює нова ОС. Якщо процесор, на який

переноситься ОС, використовує той же набір команд (можливо з деякими доповненями) і той же діапазон адрес, тоді двійкова сумісність може бути досягнута досить просто.

Набагато складніше досягти двійкової сумісності між процесорами, заснованими на різній архітектурі. Для того щоб один комп'ютер виконував програми іншого (наприклад, DOS-програму на Mac), цей комп'ютер повинен працювати з машинними командами, які йому спочатку незрозумілі.

Виходом в таких випадках є використання так званих ***прикладних середовищ***. Основну частину програми, як правило, складають виклики бібліотечних функцій. Прикладне середовище імітує бібліотечні функції, використовуючи заздалегідь написану бібліотеку функцій аналогічного призначення, а інші команди емулює кожну окремо за допомогою програми-емулятора.

3.5 Надійність, захист інформації і безпека

У жовтні 1988 року в США сталася подія, названа фахівцями найбільшим порушенням безпеки американських комп'ютерних систем з тих, що коли-небудь траплялися. 23-річний студент випускного курсу Корнельского університету Роберт Таппан Морріс запустив в комп'ютерній мережі ARPANET програму, представляючу собою різновид комп'ютерних вірусів – мережевих черв'яків. В результаті атаки був повністю або частково заблокирований ряд загальнонаціональних комп'ютерних мереж. У результаті вірус уразив більше 6200 комп'ютерних систем по всій Америці. Загальний збиток від цієї атаки був оцінений фахівцями мінімум в 100 мільйонів доларів. Р. Морріс був виключений з університету з правом повторного вступу через рік, і засуджений судом до штрафу в 270 тис. доларів і трьох місяців ув'язнення.

Важливість вирішення проблеми інформаційної безпеки нині загальновизнана, підтвердженням чому служать гучні процеси про порушення цілісності систем. Проблема забезпечення безпеки носить комплексний характер, для її вирішення потрібне поєднання законодавчих, організаційних і програмно-технічних заходів.

Технічні засоби реалізуються програмним і апаратним забезпеченням і розв'язують різні задачі з захисту і можуть бути вбудовані в операційні системи, або можуть бути реалізовані у вигляді окремих продуктів.

Система має бути захищена як від внутрішніх, так і від зовнішніх помилок, збоїв і відмов. Її дії мають бути завжди передбачуваними, а додатки не повинні мати можливості завдавати шкоди ОС. Надійність і відмовостійкість ОС передусім визначаються архітектурними рішеннями, покладеними в її основу, а також якістю її реалізації (відлагодженого коду). Крім того, важливо, чи включає ОС програмну підтримку апаратних засобів забезпечення відмовостійкості, таких, наприклад, як дискові масиви або джерела безперебійного живлення.

Із зростанням популярності систем розподілу часу виникла проблема захисту інформації. Це пов'язано зі збільшеною цінністю інформації, що обробляється комп'ютерами, а також з підвищеним рівнем загроз, існуючих при

передачі даних мережами, особливо публічними, таким як Інтернет. Багато операційних систем мають сьогодні розвинені засоби захисту інформації, засновані на шифруванні даних, аутентифікації і авторизації. У комп'ютери і ОС були вбудовані деякі інструменти загального призначення, що підтримують різні механізми захисту і забезпечують безпеку, які умовно можна ділити на дві категорії:

1. **Контроль над доступом**. Пов'язаний з регулюванням доступу користувача до системи в цілому, до її підсистем і даних, а також до різних ресурсів.
2. **Контроль над переміщенням інформації**. Регулювання потоку даних усередині системи і при їх доставці користувачеві.

Правила безпеки визначають такі властивості, як захист ресурсів одного користувача від інших і встановлення квот на ресурси для запобігання захоплення одним користувачем усіх системних ресурсів (таких як пам'ять).

Забезпечення захисту інформації від несанкціонованого доступу є обов'язковою функцією операційних систем. У більшості популярних систем гарантується міра безпеки даних, що відповідає рівню C2 системи стандартів США (Windows NT, окремі реалізації Unix та ін.).

Основи стандартів в області безпеки були закладені в документі «*Критерії оцінки надійних комп'ютерних систем»*. Цей документ, виданий в США в 1983 році національним центром комп'ютерної безпеки (NCSC – National Computer Security Center), часто називають ***Помаранчевою Книгою*** (за кольором обкладинки).

Відповідно до вимог Помаранчевої Книги безпечною вважається така система, яка «за допомогою спеціальних механізмів захисту контролює доступ до інформації таким чином, що особи, які мають тільки відповідні повноваження, або процеси, виконуються від їх імені, можуть отримати доступ на читання, запис, створення або видалення інформації».

Ієрархія рівнів безпеки, наведена в Помаранчевій Книзі, позначає нижчий рівень безпеки як D, а вищий – як A. Вважається, що такі ОС, як MS-DOS, Mac OS, OS/2, мають рівень захищеності D.

Основними властивостями, характерними для **С-**систем, є наявність підсистеми обліку подій, пов'язаних з безпекою, і вибірковий контроль доступу. Рівень **С** ділиться на 2 підрівні: рівень C1, що забезпечує захист даних від помилок користувачів, але не від дій зловмисників, і більш строгий рівень C2. На рівні C2 мають бути присутніми ***засоби секретного входу***, що забезпечують ідентифікацію користувачів шляхом введення унікального імені і пароля перед тим, як їм буде дозволений доступ до системи. ***Виборковий контроль доступу***, потрібний на цьому рівні, дозволяє власникові ресурсу визначити, хто має доступ до ресурсу і що він може з ним робити. Власник робить це шляхом надання прав доступу користувачеві або групі користувачів. ***Засоби обліку і спостереження*** (auditing) – забезпечують можливість виявити і зафіксувати важливі події, пов'язані з безпекою, або будь-які спроби створити, отримати доступ або видалити системні ресурси. ***Захист пам'яті*** – на цьому рівні система не захищена від помилок користувача, але поведінка його може бути проконтрольована за записами в журналі, залишеними засобами спостереження.

3.6 Структура побудови ОС. Архітектура ядра

Будь-яка складна система повинна мати зрозумілу і раціональну структуру, тобто розділятися на частини – модулі, що мають цілком закінчене функціональне призначення з чітко обумовленими правилами взаємодії. Ясне розуміння ролі кожного окремого модуля істотно спрощує роботу з модифікації і розвитку системи. Навпаки, складну систему без хорошої структури частіше простіше розробити наново, чим модернізувати.

Функціональна складність операційної системи неминуче призводить до складності її архітектури, під якою розуміють структурну організацію ОС на основі різних програмних модулів. До складу ОС входять виконувані і об'єктні модулі стандартних для цієї ОС форматів, бібліотеки різних типів, програмні модулі спеціального формату (наприклад, завантажувач ОС, драйвери введення-виведення), файли документації, модулі довідкової системи тощо.

Більшість сучасних операційних систем є добре структурованими модульними системами, здатними до розвитку, розширення і перенесення на нові платформи. Якої-небудь єдиної архітектури ОС не існує, але існують універсальні підходи до структуризації ОС.

3.6.1 Ядро і допоміжні модулі ОС

Найзагальнішим підходом до структуризації операційної системи є розділення усіх її модулів на дві групи:

- ядро – модулі, що виконують основні функції ОС;
- модулі, що виконують допоміжні функції ОС.

Модулі ядра виконують такі базові функції ОС, як управління процесами, пам'яттю, пристроями введення-виведення тощо. Ядро складає серцевину операційної системи, без нього ОС є повністю непрацездатною і не зможе виконати жодної зі своїх функцій.

До складу ядра входять функції, які виконують внутрісистемні задачі організації обчислювального процесу, такі як перемикання контекстів процесів, завантаження/вивантаження сторінок, обробка переривань. Ці функції недоступні для додатків.

Інший клас функцій ядра служить для підтримки додатків, створюючи для них так зване прикладне програмне середовище. Додатки можуть звертатися до ядра із запитами (системними викликами) для виконання тих або інших дій, наприклад для відкриття і читання файлу, виведення графічної інформації на дисплей, отримання системного часу тощо. Функції ядра, які можуть викликатися додатками, утворюють інтерфейс прикладного програмування – API.

Функції, що виконуються модулями ядра, є найчастіше використовуваними функціями операційної системи, тому швидкість їх виконання визначає продуктивність усієї системи в цілому. Для забезпечення високої швидкості роботи ОС усі модулі ядра або велика їх частина постійно знаходяться в оперативній пам'яті, тобто є ***резидентними***.

Ядро є рушійною силою усіх обчислювальних процесів в комп'ютерній системі, і крах ядра рівносильний краху усієї системи. Тому розробники ОС приділяють особливу увагу надійності кодів ядра, в результаті процес їх відладки може розтягуватися на багато місяців. Ядро оформляється у вигляді програмного модуля деякого спеціального формату, що відрізняється від формату додатків користувача. Термін «ядро» в різних ОС трактують по-різному. Однією з визначальних властивостей ядра є робота в привілейованому режимі.

Інші модулі ОС виконують дуже корисні, але менш обов'язкові функції. Наприклад, до таких допоміжних модулів можуть бути віднесені програми архівації даних, дефрагментації диска і тому подібні. Допоміжні модулі ОС оформляються або у вигляді додатків, або у вигляді бібліотек процедур.

Рішення про те, чи є яка-небудь програма частиною ОС чи ні, приймає виробник ОС. Серед багатьох чинників, здатних вплинути на це рішення, важливими є перспективи того, чи буде програма мати масовий попит у потенційних користувачів цієї ОС.

Окрема програма може існувати певний час як додаток користувача, а потім стати частиною ОС, або навпаки.

Допоміжні модулі ОС підрозділяються на такі групи (рис. 3.1):

- утиліти – програми, які виконують окремі задачі управління і супроводу комп'ютерної системи, такі, наприклад, як програми стискування дисків, архівації даних;
- системні обробляючі програми – текстові або графічні редактори, компілятори, компонувальники, відладчики (дебагер, англ. debugger);
- програми надання користувачеві додаткових послуг – спеціальний варіант призначеного для користувача інтерфейсу, калькулятор і навіть ігри;
- бібліотеки процедур різного призначення, що спрощують розробку додатків, наприклад бібліотека математичних функцій, функцій введення-виведення тощо.

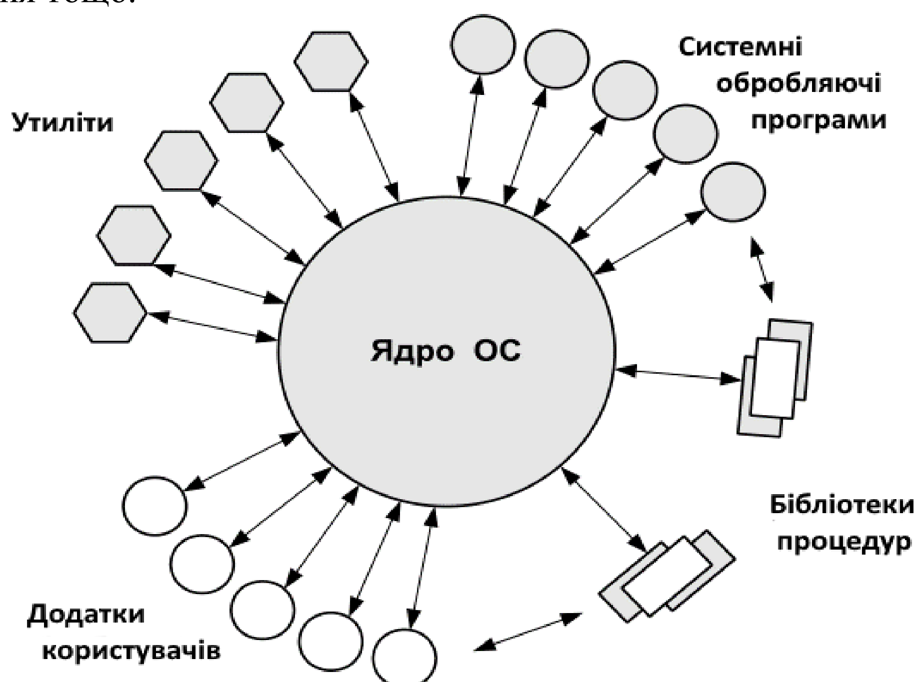


Рисунок 3.1 – Взаємодія між ядром і допоміжними модулями ОС

Для виконання своїх задач як звичайні додатки, так і утиліти, оброблювальні програми і бібліотеки ОС, звертаються до функцій ядра за допомогою системних викликів.

Модулі ОС, оформлені у вигляді утиліт, системних оброблювальних програм і бібліотек, завантажуються в оперативну пам'ять тільки на час виконання своїх функцій, тобто є ***транзитними***. Постійно в оперативній пам'яті розташовуються тільки найнеобхідніші коди ОС, що становлять її ядро. Така організація ОС економить оперативну пам'ять комп'ютера.

Розділення операційної системи на ядро і модулі-додатки забезпечує легку розширюваність ОС. Щоб додати нову високорівневу функцію, досить розробити новий додаток, і при цьому не вимагається модифікувати основні функції, що утворюють ядро системи.

3.6.2 Ядро в привілейованому режимі

Для надійного управління ходом виконання додатків операційна система повинна мати певні привілеї стосовно додатків. Інакше некоректно працюючий додаток може втрутитися в роботу ОС і, наприклад, зруйнувати частину її коду. Усі зусилля розробників операційної системи виявляться марними, якщо їх рішення втілені в незахищені від додатків модулі системи, якими б елегантними і ефективними ці рішення не були.

Операційна система повинна мати виняткові повноваження також для того, щоб грати роль арбітра в суперечці додатків за ресурси комп'ютера в мультипрограмному режимі. Жодний додаток не повинен мати можливості без відома ОС отримувати додаткову область пам'яті, займати процесор довше дозволеного операційною системою періоду часу, безпосередньо управляти спільно використовуваними зовнішніми пристроями.

Забезпечити привілеї ОС неможливо без спеціальних засобів апаратної підтримки. Апаратура комп'ютера повинна підтримувати як мінімум два режими роботи – ***режим користувача*** (user mode) і ***привілейований режим***, який також називають ***режимом ядра*** (kernel mode), або ***режимом супервізора*** (supervisor mode). Мається на увазі, що операційна система або деякі її частини працюють в привілейованому режимі, а додатки – в режимі користувача.

Оскільки ядро виконує всі основні функції ОС, то найчастіше саме ядро стає тією частиною ОС, яка працює в привілейованому режимі. Іноді ця властивість – робота в привілейованому режимі – служить основним визначенням поняття «ядро».

Підвищення стійкості операційної системи, що забезпечується переходом ядра в привілейований режим, досягається за рахунок деякого уповільнення виконання системних викликів. Системний виклик привілейованого ядра ініціює перемикання процесора з режиму користувача в привілейований, а при поверненні до додатку – перемикання з привілейованого режиму в режим користувача (рис. 3.2). В усіх типах процесорів із-за додаткової двократної затримки перемикання перехід на процедуру зі зміною режиму виконується повільніше, ніж виклик процедури без зміни режиму.

Рисунок 3.2 – Перемикання процесора з режиму користувача в привілейований режим

Додатки ставляться в підпорядковане положення за рахунок заборони виконання в режимі користувача деяких критичних команд, пов'язаних з перемиканням процесора із завдання на завдання, управлінням пристроями введення-виведення, доступом до механізмів розподілу і захисту пам'яті. Виконання деяких інструкцій в режимі користувача забороняється безумовно, тоді як інші забороняється виконувати тільки за певних умов. Очевидно, що до таких інструкцій належить інструкція переходу в привілейований режим.

Наприклад, інструкції введення-виведення можуть бути заборонені додаткам при доступі до контролера жорсткого диска, який зберігає дані, загальні для ОС і усіх додатків, але дозволені при доступі до послідовного порту, який виділений в монопольне користування для певного додатку. Важливо, що умови дозволу виконання критичних інструкцій знаходяться під повним контролем ОС і цей контроль забезпечується за рахунок набору інструкцій, безумовно заборонених для режиму користувача.

Аналогічним чином забезпечуються привілеї ОС при доступі до пам'яті. Наприклад, виконання інструкції доступу до пам'яті для додатка дозволяється, якщо інструкція звертається до області пам'яті, відведеної цьому додатку операційною системою, і забороняється при зверненні до областей пам'яті, займаних ОС або іншими додатками. Повний контроль ОС над доступом до пам'яті досягається за рахунок того, що інструкція або інструкції конфігурації механізмів захисту пам'яті дозволяється виконувати тільки в привілейованому режимі. Механізми захисту пам'яті використовуються операційною системою не лише для захисту своїх областей пам'яті від додатків, але і для захисту областей пам'яті, виділених ОС якому-небудь додатку, від інших додатків. Говорять, що кожний додаток працює у своєму ***адресному просторі***. Ця властивість дозволяє локалізувати некоректно працюючий додаток у власній області пам'яті, так що його помилки не роблять впливу на інші додатки і операційну систему.

Архітектура ОС, заснована на привілейованому ядрі і додатках, які виконуються в режимі користувача, стала, по суті, класичною. Її використовують багато популярних операційних систем, у тому числі численні версії UNIX, IBM OS/390, OS/2 і з певними модифікаціями – Windows.

У деяких випадках розробники ОС відступають від цього класичного варіанту архітектури, організовуючи роботу ядра і додатків в одному і тому ж режимі. Так, відома спеціалізована операційна система NetWare компанії Novell використовує привілейований режим процесорів Intel x86/Pentium як для роботи ядра, так і для роботи своїх специфічних додатків – завантажуваних модулів NLM. При такій побудові ОС звернення додатків до ядра виконуються швидше, оскільки немає перемикання режимів, проте при цьому відсутній надійний апаратний захист пам'яті, займаної модулями ОС, від некоректно працюючого додатку. Розробники NetWare пішли на таке потенційне зниження надійності своєї операційної системи, оскільки обмежений набір її спеціалізованих додатків дозволяє компенсувати цей архітектурний недолік за рахунок ретельної відладки кожного додатку.

В одному режимі працюють також ядро і додатки тих операційних систем, які розроблені для процесорів, що взагалі не підтримують привілейованого режиму роботи. Найпопулярнішим процесором такого типу був процесор Intel 8088/86, що послужив основою для персональних комп'ютерів компанії IBM. Операційна система MS-DOS, розроблена компанією Microsoft для цих комп'ютерів, складалася з двох модулів msdos.sys і io.sys, що складали ядро системи. До цих модулів з системними викликами зверталися командний інтерпретатор command.com, системні утиліти і додатки.

Поява в пізніших версіях процесорів Intel (починаючи з 80286) можливості роботи в привілейованому режимі не були використані розробниками MS-DOS. Ця ОС завжди працювала на процесорах цього типу в так званому ***реальному режимі***, в якому емулюється процесор 8086/88. Не слід вважати, що реальний режим є синонімом режиму користувача, а привілейований режим – його альтернативою. Реальний режим був реалізований тільки для сумісності пізніх моделей процесорів з ранньою моделлю 8086/88 і альтернативою йому є захищений режим роботи процесора з доступними всіма особливостями процесорів пізніх моделей.

3.6.3 Монолітні системи

Відмітною особливістю більшості сучасних ОС є велике ***монолітне ядро***. Ядро ОС забезпечує більшість її можливостей, включаючи планування, роботу з файловою системою, мережеві функції, роботу драйверів різних пристроїв, управління пам'яттю і багато що інше. Монолітне ядро реалізується як єдиний процес, усі елементи якого використовують один і той же адресний простір. Для побудови монолітної системи необхідно скомпілювати усі окремі процедури, а потім зв'язати їх в єдиний об'єктний файл. Тут повністю відсутнє приховування деталей реалізації – кожна процедура бачить будь-яку іншу процедуру.

У загальному випадку «структура» монолітної системи є відсутністю структури (рис. 3.3). ОС написана як набір процедур, кожна з яких може викликати інші, коли їй це треба. При використанні цієї техніки кожна процедура системи має певний інтерфейс у термінах параметрів і результатів, і кожна може викликати будь-яку іншу для виконання потрібної для неї корисної роботи.

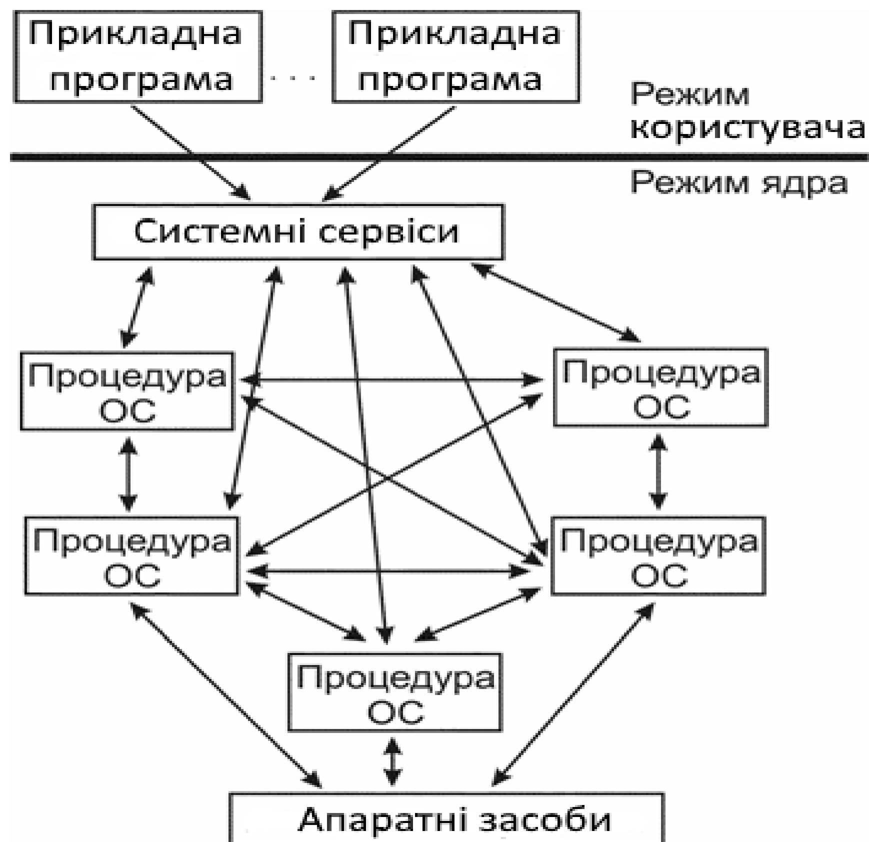


Рисунок 3.3 – Монолітна структура ОС

Така відсутність структури була несумісна з розширенням ОС, до того ж такі ОС були дуже великими і складними. Так, перша версія OS/2 була створена колективом з 5000 програмістів за 5 років і містила близько 1 млн рядків коду. Розроблена пізніше ОС Multics містила вже 20 млн. рядків коду. Прикладами ОС з монолітним ядром можуть служити ранні версії ядра UNIX. Розмір програмного коду ядра Linux в 1995 році складав 250 тисяч рядків, а в 2010 році їх число збільшилось вже до 14 мільйонів.

Проте навіть такі монолітні системи можуть бути трохи структурованими. При зверненні до системних викликів, підтримуваних ОС, параметри поміщаються в строго певні місця, такі, як регістри або стек, а потім виконується спеціальна команда переривання, відома як ***виклик ядра*** або ***виклик супервізора***. Ця команда перемикає машину з режиму користувача в режим ядра, що називається також режимом супервізора, і передає управління ОС. Потім ОС перевіряє параметри виклику для того, щоб визначити, який системний виклик має бути виконаний. Після цього ОС індексує таблицю, що містить посилання на процедури, і викликає відповідну процедуру. Така організація ОС припускає таку структуру:

1. Головна програма, яка викликає необхідні сервісні процедури.
2. Набір сервісних процедур, що реалізовують системні виклики.
3. Набір утиліт, обслуговуючих сервісні процедури.

У цій моделі для кожного системного виклику є одна сервісна процедура. Утиліти виконують функції, які потрібні декільком сервісним процедурам. Це ділення процедур на три шари показане на рис. 3.4.

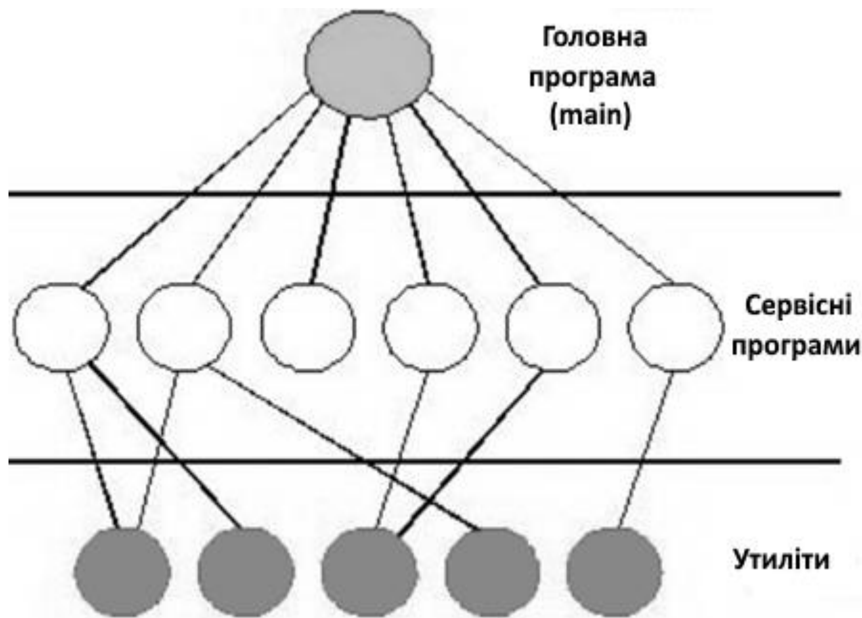


Рисунок 3.4 – Проста структуризація монолітної ОС

3.6.4 Багаторівневі (багатошарові) системи

Узагальненням попереднього підходу є організація ОС як ієрархії рівнів. Рівні утворюються групами функцій ОС – файлова система, управління процесами і пристроями і тому подібне. Кожен рівень може взаємодіяти тільки зі своїм безпосереднім сусідом, який знаходиться вище або нижче.

Обчислювальну систему, працюючу під управлінням ОС на основі ядра, можна розглядати як систему, що складається з трьох основних ієрархічно розташованих шарів: нижній шар утворює апаратуру, проміжний – ядро, а утиліти, оброблювальні програми і додатки, складають верхній шар системи (рис. 3.5). Шарову структуру обчислювальної системи зручно зображати у вигляді системи концентричних кіл, ілюструючи той факт, що кожен шар може взаємодіяти тільки з суміжними шарами. При такій організації ОС додатки не можуть безпосередньо взаємодіяти з апаратурою, а тільки через шар ядра.

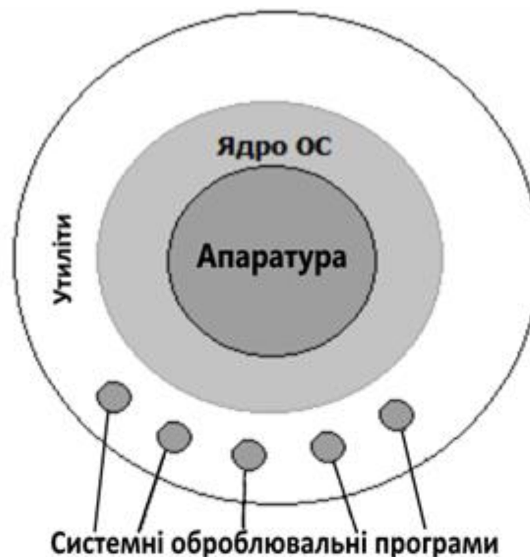


Рисунок 3.5 – Тришарова схема обчислювальної системи

Багатошаровий підхід є універсальним і ефективним способом декомпозиції складних систем будь-якого типу, у тому числі і програмних. Відповідно до цього підходу система складається з ієрархії шарів. Кожен шар обслуговує вищерозміщений шар, виконуючи для нього деякий набір функцій, які утворюють міжшаровий інтерфейс. На основі функцій шару, що пролягає нижче, наступний (вгору за ієрархією) шар будує свої функції – складніші і потужніші, які, у свою чергу, виявляються примітивами для створення ще потужніших функцій вищерозміщеного шару. Суворі правила стосуються тільки взаємодії між шарами системи, а між модулями усередині шару зв'язки можуть бути довільними. Окремий модуль може виконати свою роботу або самостійно, або звернутися до іншого модуля свого шару, або звернутися за допомогою до шару, що пролягає нижче, через міжшаровий інтерфейс.

Першою системою, побудованою таким чином, була пакетна система **THE** (Technishe Hogeschool Eindhoven) Multiprogramming System, яку створив Е. Дейкстра і його студенти в 1968 році (рис. 3.6). Система мала 6 рівнів (шарів).

Рівень 0 займався розподілом часу процесора, перемикаючи процеси за перериванням або після закінчення часу.

Рівень 1 управляв пам'яттю – розподіляв оперативну пам'ять і простір на магнітному барабані для тих частин процесів (сторінок), для яких не було місця в ОП, тобто шар 1 виконував функції віртуальної пам'яті.

Рисунок 3.6 – Багатошарова операційна система THE

Рівень 2 управляв зв'язком між консоллю оператора і процесами. За допомогою цього рівня кожен процес мав свою власну консоль оператора.

Рівень 3 управляв пристроями введення-виведення і буферизував потоки інформації до них і від них. За допомогою рівня 3 кожен процес замість того, щоб працювати з конкретними пристроями, з їх різноманітними особливостями, звертався до абстрактних пристроїв введення-виведення.

На рівні 4 працювали призначені для користувача програми, яким не потрібно було піклуватися ні про процеси, ні про пам'ять, ні про управління пристроями введення-виведення.

На рівні 5 розміщувався процес системного оператора.

Оскільки ядро є складним багатофункціональним комплексом, то багатошаровий підхід поширюється і на структуру ядра. Ядро може складатися з таких шарів (рис. 3.7).

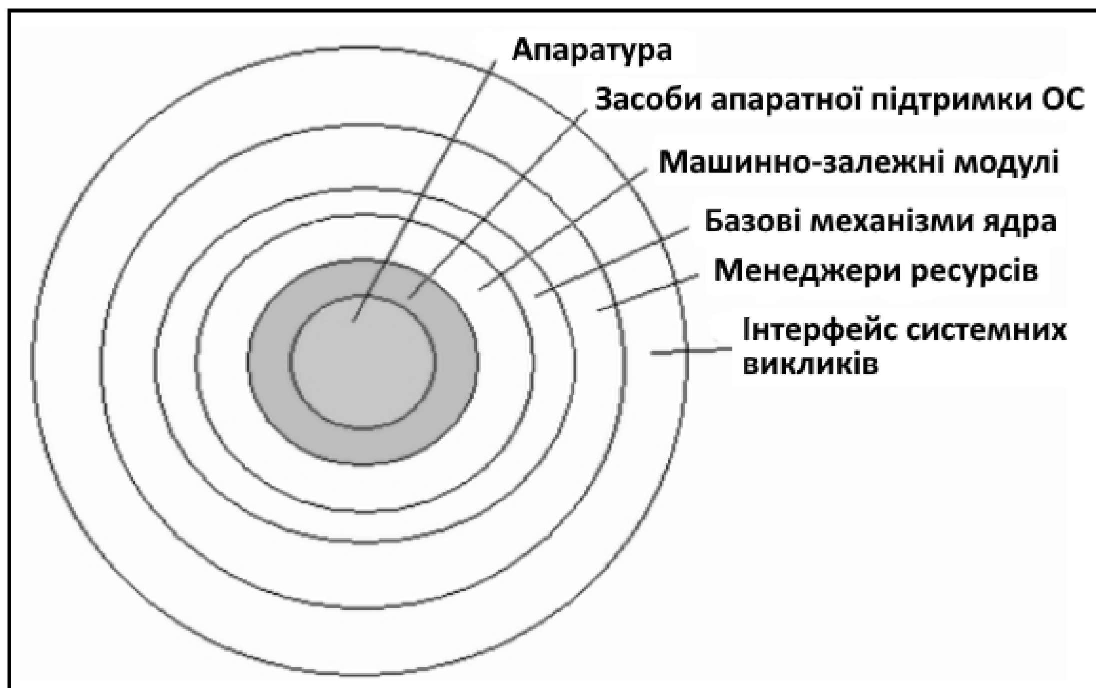


Рисунок 3.7 – Багатошарова структура ядра ОС

Засоби апаратної підтримки ОС. Досі про операційну систему говорилося як про комплекс програм, але, взагалі кажучи, частина функцій ОС може виконуватися і апаратними засобами. Тому іноді можна зустріти визначення операційної системи як сукупності програмних і апаратних засобів, що й показано на рис. 3.7. До операційної системи відносять не всі апаратні пристрої комп'ютера, а тільки засоби апаратної підтримки ОС, тобто ті, які прямо беруть участь в організації обчислювальних процесів: засоби підтримки привілейованого режиму, систему переривань, засоби перемикання контекстів процесів, засоби захисту областей пам'яті тощо.

Машинно-залежні компоненти ОС. Цей шар утворюють програмні модулі, в яких відбивається специфіка апаратної платформи комп'ютера. В ідеалі цей шар повністю екранує вищерозміщені шари ядра від особливостей апаратури. Це дозволяє розробляти вищерозміщені шари на основі машинно-незалежних модулів, існуючих в єдиному екземплярі для усіх типів апаратних платформ, підтримуваних цією ОС. Прикладом екрануючого шару може служити шар HAL операційної системи Windows NT.

Базові механізми ядра. Цей шар виконує найпримітивніші операції ядра, такі як програмне перемикання контекстів процесів, диспетчеризацію переривань, переміщення сторінок з пам'яті на диск і назад тощо. Модулі цього шару не приймають рішень про розподіл ресурсів – вони тільки відпрацьовують прийняті «вгорі» рішення, що і дає привід називати їх виконавчими механізмами для модулів верхніх шарів. Наприклад, рішення про те, що в даний момент треба перервати виконання поточного процесу *A* і почати виконання процесу *B*,

приймається менеджером процесів на вищерозміщеному шарі, а шару базових механізмів передається тільки директива про те, що треба виконати перемикання з контексту поточного процесу на контекст процесу *B*.

Менеджери ресурсів. Цей шар складається з потужних функціональних модулів, що реалізовують стратегічні завдання з управління основними ресурсами обчислювальної системи. На цьому шарі працюють менеджери (які називаються також диспетчерами) процесів, введення-виведення, файлової системи і оперативної пам'яті.

Розбиття на менеджери може бути і дещо іншим. Наприклад, менеджер файлової системи іноді об'єднують з менеджером введення-виведення, а функції управління доступом користувачів до системи в цілому і її окремим об'єктам доручають окремому менеджерові безпеки. Кожен з менеджерів веде облік вільних і використовуваних ресурсів певного типу і планує їх розподіл відповідно до запитів додатків.

Менеджер віртуальної пам'яті управляє переміщенням сторінок з оперативної пам'яті на диск і назад. Менеджер повинен відстежувати інтенсивність звернень до сторінок, час перебування їх в пам'яті, стану процесів, що використовують дані, і багато інших параметрів, на підставі яких він час від часу приймає рішення про те, які сторінки необхідно вивантажити і які – завантажити.

Для виконання прийнятих рішень менеджер звертається до шару базових механізмів, що пролягає нижче, із запитами про завантаження (вивантаженню) конкретних сторінок. Усередині шару менеджерів існують тісні взаємні зв'язки, що відбивають той факт, що для виконання процесу потрібний доступ одночасно до декількох ресурсів – процесора, області пам'яті, можливо, до певного файлу або пристроїв введення-виведення. Наприклад, при створенні процесу менеджер процесів звертається до менеджера пам'яті, який повинен виділити процесу певну область пам'яті для його кодів і даних.

Інтерфейс системних викликів. Цей шар є самим верхнім шаром ядра і взаємодіє безпосередньо з додатками і системними утилітами, утворюючи прикладний програмний інтерфейс операційної системи. Функції API, обслуговуючі системні виклики, надають доступ до ресурсів системи в зручній і компактній формі, без вказівки деталей їх фізичного розташування. Для здійснення таких комплексних дій системні виклики звертаються за допомогою до функцій шару менеджерів ресурсів, причому для виконання одного системного виклику може знадобитися декілька таких звернень.

Наведене розбиття ядра ОС на шари є досить умовним. У реальній системі кількість шарів і розподіл функцій між ними може бути і іншим. У системах, призначених для апаратних платформ одного типу, наприклад ОС NetWare, шар машинно-залежних модулів не виділяється, зливаючись з шаром базових механізмів і, частково, з шаром менеджерів ресурсів. Не завжди оформляються в окремий шар базові механізми. У цьому випадку менеджери ресурсів не лише планують використання ресурсів, але і самостійно реалізують свої плани.

Можлива і протилежна картина, коли ядро складається з більшої кількості шарів. Наприклад, менеджери ресурсів, складаючи певний шар ядра, у свою

чергу, можуть мати багатошарову структуру. Передусім це стосується до менеджера введення-виведення, нижній шар якого складають драйвери пристроїв, наприклад драйвер жорсткого диска або драйвер мережевого адаптера, а верхні шари – драйвери файлових систем або протоколів мережевих служб, що мають справу з логічною організацією інформації.

Спосіб взаємодії шарів в реальній ОС також може відхилятися від описаної вище схеми. Для прискорення роботи ядра в деяких випадках відбувається безпосереднє звернення з верхнього шару до функцій нижніх шарів, оминаючи проміжні. На багатьох апаратних платформах для реалізації системного виклику використовується інструкція програмного переривання. Цим додаток фактично викликає модуль первинної обробки переривань, який знаходиться в шарі базових механізмів, а вже цей модуль викликає потрібну функцію з шару системних викликів. Самі функції системних викликів також іноді порушують субординацію ієрархічних шарів, звертаючись прямо до базових механізмів ядра.

Вибір кількості шарів ядра є відповідальною і складною справою. Збільшення числа шарів веде до деякого уповільнення роботи ядра за рахунок додаткових накладних витрат на міжшарову взаємодію, а зменшення числа шарів погіршує розширюваність і логічність системи. ОС, що пройшли довгий шлях еволюційного розвитку, наприклад, багато версій UNIX, мають невпорядковане ядро з невеликим числом чітко виділених шарів (рис. 3.8).

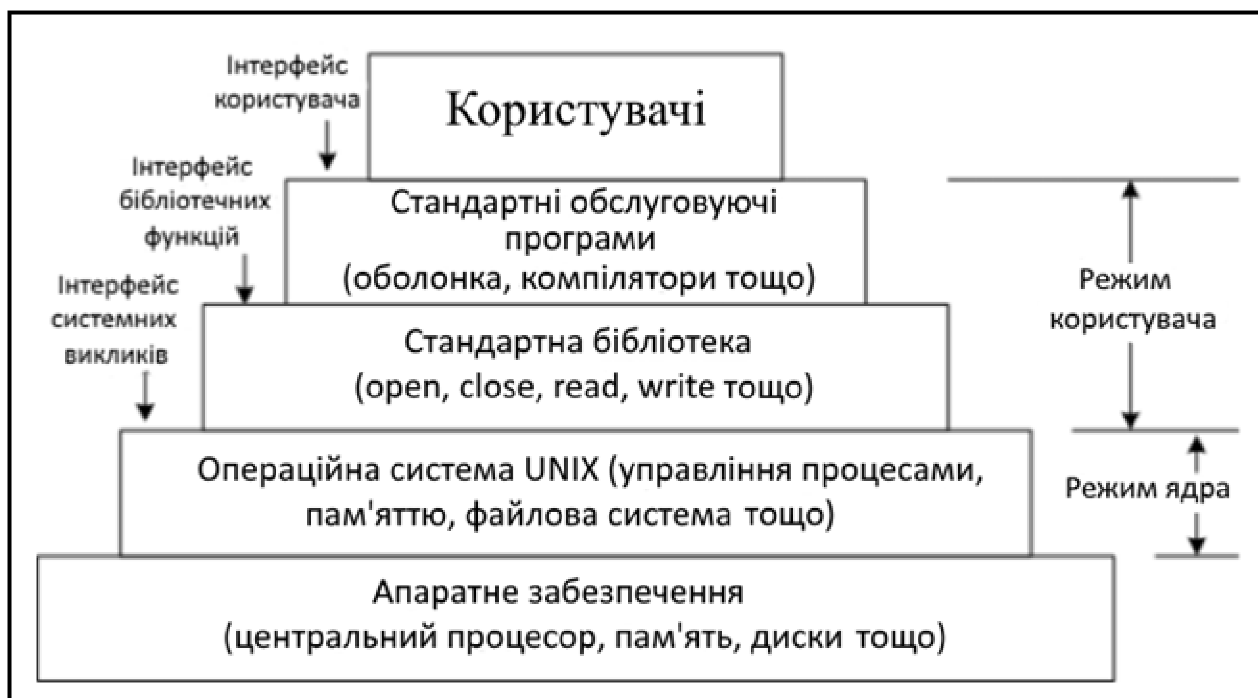


Рисунок 3.8 – Структура ОС UNIX

У порівняно «молодих» операційних систем, таких як Windows NT, ядро розділене на більше число шарів і їх взаємодія формалізована набагато краще (рис. 3.9).

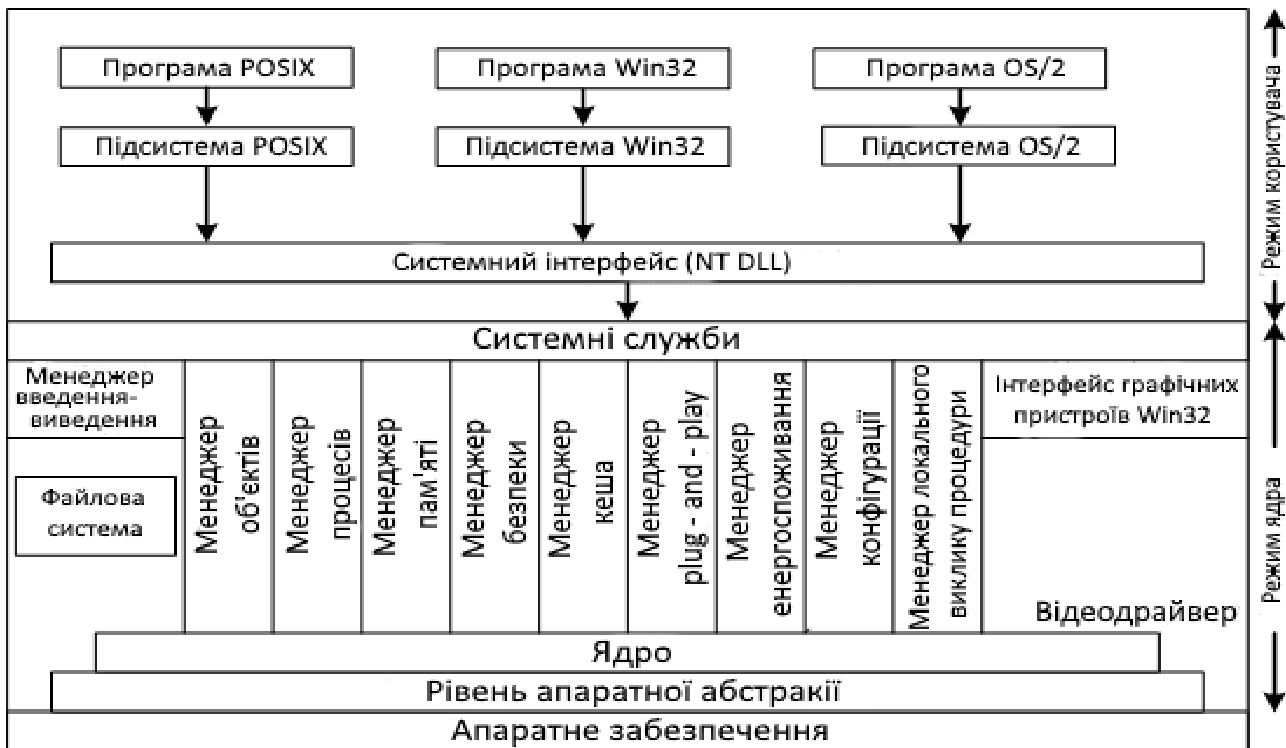


Рисунок 3.9 – Структура Windows 2000

Хоча такий структурний підхід на практиці працював непогано (висока продуктивність), сьогодні він все більше сприймається монолітним. У системах, що мають багаторівневу структуру, було нелегко видалити один шар і замінити його іншим в силу множинності і розмитості інтерфейсів між шарами. Додавання нових функцій і зміна існуючих вимагали хорошого знання операційної системи і багато часу (великий код ядра, і як наслідок великий вміст помилок).

Коли стало ясно, що операційні системи живуть довго і повинні мати можливості розвитку і розширення, монолітний і багатошаровий підходи стали давати тріщину, і на зміну їм прийшла модель клієнт-сервер і тісно пов'язана з нею концепція мікроядра.

3.6.5 Модель клієнт-сервер і мікроядро

Модель ***клієнт-сервер*** – це ще один підхід до структуризації ОС. У широкому сенсі модель клієнт-сервер припускає наявність програмного компонента-споживача якого-небудь сервісу – ***клієнта***, і програмного компонента-постачальника цього сервісу – ***сервера***. Взаємодія між клієнтом і сервером стандартизується, так що сервер може обслуговувати клієнтів, реалізованих різними способами і, можливо, різними виробниками. При цьому головною вимогою є те, щоб вони просили послуги сервера зрозумілим йому способом. Ініціатором обміну є клієнт, який посилає запит на обслуговування серверу, що знаходиться в стані очікування запиту.

Ще в деяких ранніх ОС частина функцій реалізувалася в режимі користувача, оскільки системний виклик не був потрібний. Системний виклик

виконується повільніше, ніж виклик функції, реалізованої в режимі користувача, оскільки процесор двічі перемикається між режимами.

До складу мікроядра входять машинно-залежні модулі, а також модулі, що виконують базові (але не всі) функції ядра з управління процесами, обробки переривань, управління віртуальною пам'яттю, пересилання повідомлень і управління пристроями введення-виведення, пов'язані із завантаженням або читанням регістрів пристроїв. Набір функцій мікроядра відповідає функціям шару базових механізмів звичайного ядра. Такі функції операційної системи важко, якщо не неможливо, виконати в просторі користувача.

Усі інші більш високорівневі функції ядра оформляються у вигляді додатків (процеси-сервери), працюючих в режимі користувача. Такий підхід дозволяє розділити задачу розробки ОС на розробку ядра і розробку серверів. Сервери можна Налагоджувати для вимог конкретних додатків або середовища. Виділення в структурі системи мікроядра спрощує реалізацію системи, забезпечує її гнучкість (рис. 3.10).

Стосовно структуризації ОС ідея полягає в розбитті її на декілька ***процесів-серверів***, кожен з яких виконує окремий набір сервісних функцій – наприклад, управління пам'яттю, створення або планування процесів. Кожен сервер виконується в призначеному для користувача режимі.

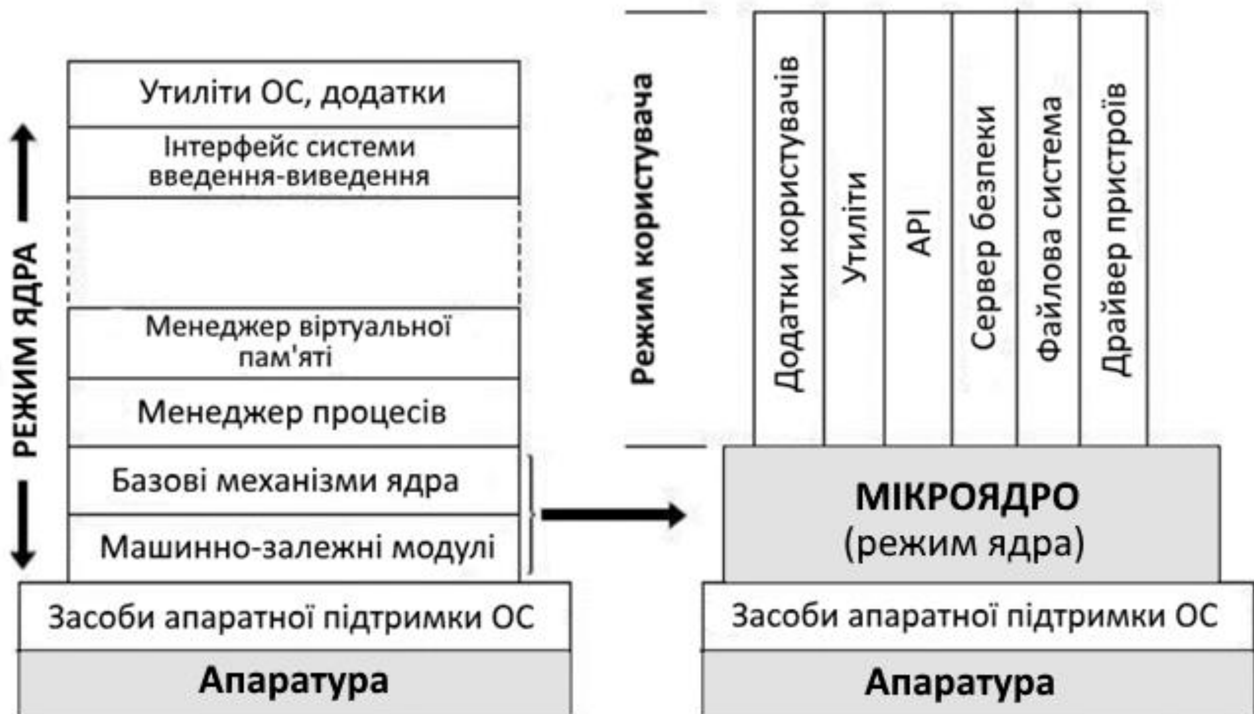


Рисунок 3.10 – Перехід до мікроядерної архітектури

Клієнт, яким може бути або інший компонент ОС, або прикладна програма, просить сервіс, посилаючи повідомлення на сервер. Ядро ОС (що називається тут ***мікроядром***), працюючи в привілейованому режимі, доставляє повідомлення потрібного сервера, сервер виконує операцію, після чого ядро повертає результати клієнтові за допомогою іншого повідомлення (рис. 3.11).

Рисунок 3.11 – Структура ОС клієнт-сервер

Мікроядерна архітектура є альтернативою класичному способу побудови операційної системи. Під класичною архітектурою в даному випадку розуміється розглянута вище структурна організація ОС, відповідно до якої всі основні функції операційної системи, що становлять багатошарове ядро, виконуються в привілейованому режимі.

Суть мікроядерної архітектури полягає в такому. У привілейованому режимі залишається працювати тільки дуже невелика частина ОС, що називається мікроядром. Мікроядро захищене від інших частин ОС і додатків.

Підхід з використанням мікроядра замінив вертикальний розподіл функцій операційної системи на горизонтальний. Компоненти, мікроядра, що лежать вище, хоча і використовують повідомлення, що пересилаються через мікроядро, взаємодіють один з одним безпосередньо. Мікроядро грає роль регулювальника. Воно перевіряє повідомлення, пересилає їх між серверами і клієнтами, і надає доступ до апаратури.

Схема зміни режимів при виконанні системного виклику в ОС з мікроядерною архітектурою виглядає, як показана на рис. 3.12. З рисунка видно, що при класичній організації ОС (рис. 3.12, а) виконання системного виклику супроводжується двома перемиканнями режимів (*2 t*). При мікроядерній організації (рис. 3.12, б) – чотирма (*4 t*). Отже, ОС з мікроядерною архітектурою за інших рівних умов завжди буде менш продуктивною, чим ОС з класичним ядром.

Мікроядро реалізує важливі функції, що лежать в основі операційної системи. Це базис для менш істотних системних служб і додатків. Саме питання про те, які з системних функцій вважати **несуттєвими**, і, відповідно, не включати їх до складу ядра, є предметом суперечки серед прибічників ідеї мікроядра. У загальному випадку, підсистеми, що були традиційно невід'ємними частинами ОС, – файлові системи, управління вікнами і забезпечення безпеки – стають периферійними модулями, що взаємодіють з ядром і один з одним, і працюють в режимі користувача.

Рисунок 3.12 – Обробка системного виклику в микроядерній архітектурі

Головний принцип розділення роботи між мікроядром і модулями, що оточують його, – включати в мікроядро тільки ті функції, яким абсолютно необхідно виконуватися в режимі супервізора і в привілейованому просторі. Під цим маються на увазі машинно-залежні програми, деякі функції управління процесами, обробка переривань, підтримка пересилки повідомлень, деякі функції управління пристроями введення-виведення, пов'язані із завантаженням команд в регістри пристроїв. Ці функції операційної системи важко, якщо не неможливо, виконати програмам, працюючим в просторі користувача.

Тут важливо зробити відмінність. Запуск процесу або потоку вимагає доступу до апаратури, так що за логікою – це функція ядра. Але ядру все одно, який з потоків запускати, тому рішення про пріоритети потоків і дисципліну постановки в чергу може приймати працюючий поза ядром планувальник.

Окрім вже представлених міркувань, переміщення планувальника на призначений для користувача рівень може знадобитися для чисто комерційних цілей. Деякі виробники ОС планують ліцензувати своє мікроядро іншим постачальникам, яким може потрібно замінити початковий планувальник на інший, такий, що підтримує, наприклад, планування в завданнях реального часу або що реалізовує якийсь спеціальний алгоритм планування.

Як і управління процесами, управління пам'яттю може розподілятися між мікроядром і сервером, працюючим в режимі користувача.

Драйвери пристроїв також можуть розташовуватися як усередині ядра, так і поза ним. При розміщенні драйверів пристроїв поза мікроядром для забезпечення можливості дозволу і заборони переривань, частина програми драйвера повинна виконуватися в просторі ядра. Відділення драйверів пристроїв від ядра робить можливою динамічну конфігурацію ОС.

Окрім динамічної конфігурації, є і інші причини розглядати драйвери пристроїв в якості процесів режиму користувача. СУБД, наприклад, може мати свій драйвер, оптимізований під конкретний вид доступу до диска, але його не можна буде підключити, якщо драйвери будуть розташовані в ядрі. Цей підхід також сприяє переносимості системи, оскільки функції драйверів пристроїв можуть бути в багатьох випадках абстраговані від апаратної частини.

Нині саме операційні системи, побудовані з використанням моделі клієнт-сервер і концепції мікроядра, найбільшою мірою задовольняють вимогам, що пред'являються до сучасних ОС.

Однаковий інтерфейс. Використання мікроядра припускає однаковий інтерфейс запитів, генерованих процесами. Процесам не треба відрізняти служби, що виконуються на рівні ядра і на призначеному для користувача рівні, оскільки доступ до усіх цих служб здійснюється тільки за допомогою передачі повідомлень.

Можливість розширення ОС. Архітектура мікроядра сприяє розширюваності ОС, дозволяючи додавати в них нові сервісні, а також забезпечувати нові множинні сервіси в одній і тій же функціональній області. Складність монолітних операційних систем, що збільшується, зробила важким, якщо взагалі можливим, внесення змін до ОС з гарантією надійності її подальшої роботи. Обмежений набір чітко певних інтерфейсів мікроядра відкриває шлях до впорядкованого зростання і еволюції ОС.

Переносимість. В архітектурі з мікроядром спеціально призначений для конкретного процесора код, або велика його частина, знаходиться в мікроядрі. Тому зміни, необхідні для перенесення системи на новий процесор, зводяться до мінімуму і мають тенденцію до розміщення в окремих логічних групах.

Надійність. Хоча модульність допомагає підвищити надійність роботи системи, архітектура з мікроядром дає ще вагоміші переваги. Маленьке мікроядро можна ретельно протестувати. Використання в ньому невеликої кількості інтерфейсів прикладного програмування підвищує шанси на те, що сервіси ОС, які працюють поза ядром, будуть реалізовані за допомогою якісного коду. Системний програміст має у своєму розпорядженні обмежену кількість інтерфейсів і обмежені способи організації взаємодії, тому він не зможе негативно вплинути на інші системні компоненти.

Кожен сервер виконується у вигляді окремого процесу у своїй власній області пам'яті, і таким чином захищений від інших процесів і від інших серверів операційної системи. Більше того, оскільки сервери виконуються в просторі користувача, вони не мають безпосереднього доступу до апаратури і не можуть модифікувати пам'ять, в якій зберігається управляюча програма. І якщо окремий сервер може потерпіти крах, то він може бути перезапущений без зупинки або ушкодження іншої частини ОС.

Обробка апаратних переривань. В архітектурі мікроядра є можливість обробляти апаратні переривання аналогічно повідомленням, а також включати в адресний простір порти введення-виведення. Таке мікроядро може розпізнавати переривання, але не обробляти їх. Замість цього воно генерує повідомлення процесу (сервісу), який виконується на рівні користувача і пов'язаному з цим перериванням. Таким чином, коли переривання дозволене, з ним зіставляється процес на рівні користувача, і таке відображення підтримується ядром.

Розподілення обчислень. Ця модель також добре підходить для розподілених обчислень, оскільки окремі сервери можуть працювати на різних процесорах мультипроцесорного комп'ютера або навіть на різних комп'ютерах, оскільки використовує механізми, аналогічні мережевим: взаємодія клієнтів і

серверів шляхом обміну повідомленнями. При отриманні повідомлення від процесу мікроядро може обробити його самостійно або переслати іншому процесу, оскільки мікроядру все одно, чи прийшло повідомлення від локального або віддаленого процесу.

Прикладне середовище. Відповідно до мікроядерної архітектури всі функції ОС реалізуються мікроядром і серверами в режиме користувача. Важливо, що кожне прикладне середовище оформляється у вигляді окремого сервера і не включає базових механізмів (рис. 3. 13).

Додатки, використовуючи API, поводяться з системними викликами до відповідного прикладного середовища через мікроядро. Прикладне середовище обробляє запит, виконує його (можливо, за допомогою базових функцій мікроядра) і посилає додатку результат. У ході виконання запиту прикладному середовищу доводиться, у свою чергу, звертатися до базових механізмів ОС, мікроядра та до інших серверів ОС.

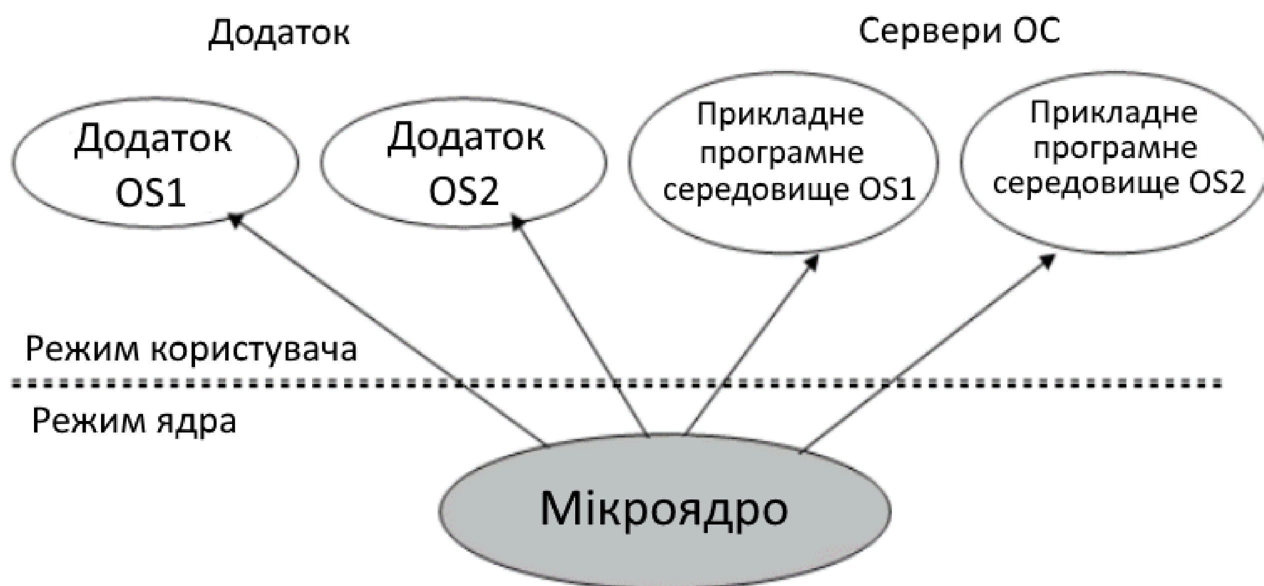


Рисунок 3. 13 – Мікроядерний підхід до реалізації багатьох прикладних середовищ

Головним недоліком мікроядерного підходу є зниження продуктивності. Системний виклик виконується повільніше, ніж виклик функції, реалізованої в режимі користувача, оскільки замість двох перемикань режиму процесора в разі системного виклику при класичній організації ОС здійснюється чотири (два – під час обміну між клієнтом і мікроядром, два – між сервером і ядром).

Але все таки цей недолік є більше теоретичним, оскільки на практиці продуктивність і надійність ядра залежить безпосередньо від якості його реалізації.

Серйозність цього недоліку добре ілюструє історія розвитку Windows NT. У версіях 3.1 і 3.5 диспетчер вікон, графічна бібліотека і високорівневі драйвери графічних пристроїв входили до складу сервера призначеного для користувача режиму, і виклик функцій цих модулів здійснювався відповідно до мікроядерної схеми. Проте дуже скоро розробники Windows NT зрозуміли, що такий механізм звернень до часто використовуваних функцій графічного інтерфейсу істотно

уповільнює роботу додатків і робить цю операційну систему уразливою в умовах гострої конкуренції. В результаті у версію Windows NT 4.0 були внесені істотні зміни – усі перелічені вище модулі були перенесені в ядро, що віддалило цю ОС від ідеальної мікроядерної архітектури, та зате різко підвищило її продуктивність.

Існує ще одна проблема, з якою стикаються розробники операційної системи, які вирішили застосувати мікроядерний підхід, – що включати в мікроядро, а що виносити в призначений для користувача простір. В ідеальному випадку мікроядро може складатися тільки із засобів передачі повідомлень, засобів взаємодії з апаратурою, у тому числі засобів доступу до механізмів привілейованого захисту. Проте багато розробників не завжди жорстко дотримуються принципу мінімізації функцій ядра, часто жертвуючи цим заради підвищення продуктивності.

Для можливості уявлення про розміри мікроядер операційних систем у ряді джерел наводяться такі дані:

- типове мікроядро першого покоління – 300 Кб коду і 140 інтерфейсів системних викликів;
- мікроядро ОС L4 (друге покоління) – 12 Кб коду і 7 інтерфейсів системних викликів;
- мікроядро UNIX-подібної POSIX-сумісної, 32-бітової, багатозадачної, розрахованої на багато користувачів, високопродуктивної операційної системи реального часу QNX фірми Quantum Software systems (Канада) займає декілька кілобайт пам'яті і забезпечує мінімальний набір функцій.

У сучасних операційних системах відрізняють такі види ядер.

Наноядро. Украй спрощене і мінімальне ядро, виконує лише одне завдання – обробку апаратних переривань, генерованих облаштуваннями комп'ютера. Після обробки посилає інформацію про результати обробки вищерозміщеному програмному забезпеченню.

Мікроядро надає тільки елементарні функції управління процесами і мінімальний набір абстракцій для роботи з устаткуванням. Велика частина роботи здійснюється за допомогою спеціальних призначених для користувача процесів, що називаються сервісами.

Екзоядро надає лише набір сервісів для взаємодії між додатками, а також необхідний мінімум функцій, пов'язаних із захистом, виділенням і вивільненням ресурсів, контролем прав доступу тощо.

Монолітне ядро надає широкий набір абстракцій устаткування. Усі частини ядра працюють в одному адресному просторі. Монолітне ядро вимагає перекомпіляції при зміні складу устаткування. Компоненти операційної системи є не самостійними модулями, а складовими частинами однієї програми. Монолітне ядро продуктивніше, ніж мікроядро, оскільки працює як один великий процес.

Модульне ядро – сучасна, вдосконалена модифікація архітектури мікроядра. На відміну від «класичного» монолітного ядра, модульні ядра не вимагають повної перекомпіляції ядра при зміні складу апаратного забезпечення комп'ютера. Замість цього вони надають той або інший механізм підвантаження

модулів, що підтримують те або інше апаратне забезпечення (наприклад, драйверів).

Гібридне ядро – це модифіковані мікроядра, що дозволяють для прискорення роботи запускати «несуттєві» частини в просторі ядра. Прикладом гібридного підходу може служити можливість запуску операційної системи з монолітним ядром під управлінням мікроядра. Так влаштовані 4.4BSD і MkLinux, засновані на мікроядрі Mach.

Ядро Windows NT. Найтісніше елементи мікроядерної архітектури і елементи монолітного ядра переплетені в ядрі Windows NT. Хоча Windows NT часто називають мікроядерною операційною системою, це не зовсім так. Мікроядро NT занадто велике (більше 1 Мб), щоб носити приставку «мікро». Компоненти ядра Windows NT розташовуються в пам'яті, що витісняється, і взаємодіють шляхом передачі повідомлень, як і належить в мікроядерних операційних системах. В той же час усі компоненти ядра працюють в одному адресному просторі і активно використовують загальні структури даних, що властиво операційним системам з монолітним ядром.

3.6.6 Об'єктно-орієнтований підхід

Хоча технологія мікроядер і заклала основи модульних систем, вона не змогла в повній мірі забезпечити можливості розширення систем. Нині цій меті найбільше відповідає об'єктно-орієнтований підхід, при якому кожен програмний компонент є функціонально ізольованим від інших.

Основним поняттям цього підходу є «об'єкт». ***Об'єкт*** – це одиниця програм і даних, що взаємодіє з іншими об'єктам за допомогою прийому і передачі повідомлень. Об'єкт може бути представленням як прикладної програми або документу, так і деяких абстракцій – процесу, події.

Програми (функції) об'єкта визначають перелік дій, які можуть бути виконані над даними цього об'єкта. Об'єкт-клієнт може звернутися до іншого об'єкта, пославши сполучення із запитом на виконання якої-небудь функції об'єкта-сервера.

Об'єкти можуть описувати сутності, які вони представляють, з різною мірою деталізації. Для забезпечення спадкоємності при переході до детальнішого опису розробникам пропонується механізм ***спадкоємства*** властивостей вже існуючих об'єктів, тобто механізм, що дозволяє породжувати конкретніші об'єкти із загальних. Наприклад, за наявності об'єкту «текстовий документ» розробник може легко створити об'єкт «текстовий документ у форматі Word», додавши відповідну властивість до базового об'єкта. Механізм спадкоємства дозволяє створити ієрархію об'єктів, в якій кожен об'єкт нижчого рівня набуває всіх властивостей свого предка (пробатька).

Внутрішня структура даних об'єкту прихована від спостереження. Не можна довільно змінювати дані об'єкта. Для того щоб отримати дані з об'єкта або помістити дані в об'єкт, необхідно викликати відповідні об'єктні функції. Це ізолює об'єкт від того коду, який використовує його. Розробник може звертатися до функцій інших об'єктів, або будувати нові об'єкти шляхом наслідування

властивостей інших об'єктів, нічого не знаючи про те, як вони сконструйовані. Ця властивість називається ***інкапсуляцією***.

Таким чином, об'єкт з'являється для зовнішнього світу у вигляді «чорного ящика» з певним інтерфейсом. З точки зору розробника, що використовує об'єкт, поки зовнішня реакція об'єкта залишається без змін, не мають значення ніякі зміни у внутрішній реалізації. Це дає можливість легко замінювати одну реалізацію об'єкта іншою. наприклад, у разі зміни апаратних засобів. З іншого боку, здатність об'єктів з'являтися у вигляді «чорного ящика» дозволяє упаковувати в них і представляти у вигляді об'єктів уже існуючі додатки, нічого в них не змінюючи.

Повністю об'єктно-орієнтовані операційні системи дуже привабливі для системних програмістів, оскільки, використовуючи об'єкти системного рівня, програмісти зможуть залізати вглиб операційних систем для пристосування їх до своїх потреб, не порушуючи цілісність системи. Об'єктно-орієнтований підхід є однією з найперспективніших тенденцій в конструюванні програмного забезпечення. Він був прийнятий на озброєння багатьма відомими фірмами, такими як Microsoft, Apple, IBM, Novell/USL (UNIX Systems Laboratories) і Sun Microsystems – усі вони розгорнули свої операційні системи в цьому напрямі.

3.7 Множинні прикладні середовища

При реалізації множинних прикладних середовищ розробники стикаються з суперечливими вимогами. З одного боку, завданням кожного прикладного середовища є виконання програми по можливості так, як коли б вона виконувалася на «рідній» ОС. Але потреби цих програм можуть входити в конфлікт з конструкцією сучасної операційної системи. Спеціалізовані драйвери пристроїв можуть суперечити вимогам безпеки. Можуть конфліктувати схеми управління пам'яттю і віконні системи. Але найбільшою потенційною проблемою є продуктивність – прикладне середовище повинне виконувати програми з прийнятною швидкістю.

Цій вимозі не відповідають системи емуляції. Для скорочення часу на виконання «чужих» програм прикладні середовища використовують імітацію програм на рівні бібліотек. Ефективність цього підходу пов'язана з тим, що більшість сьогоднішніх програм виконуються під управлінням GUI (графічних інтерфейсів користувача) типу Windows, Mac або UNIX Motif. При цьому додатки витрачають велику частину часу, роблячи деякі добре передбачувані речі. Вони безперервно виконують виклики бібліотек GUI для маніпулювання вікнами і для інших пов'язаних з GUI дій. І це те, що дозволяє прикладним середовищам відшкодувати час, витрачений на емуляцію команди за командою.

Модульність операційних систем нового покоління дозволяє набагато легше реалізувати підтримку множинних прикладних середовищ. На відміну від старих операційних систем, що складаються з одного великого блоку для усіх практичних застосувань, розбитого довільним чином на частини, нові системи є модульними, з чітко певними інтерфейсами між складовими. Це робить

створення додаткових модулів, що об'єднують емуляцію процесора і трансляцію бібліотек, значно простішим.

До вдосконалених операційних системам, що явно містять засоби множинних прикладних середовищ, належать: IBM OS/2 2.x, Microsoft Windows NT і версії UNIX від Sun Microsystems, IBM і Hewlett-Packard.

3.8 Концепція віртуальних машин

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

Уперше концепція ВМ була реалізована в 70-і роки в ОС VM фірми IBM. У СРСР варіант цієї системи (VM/370) був широко поширений в 80-і роки, і мав назву Системи Віртуальних Машин (СВМ) ЄС.

Ядро системи, що називається ***монітором віртуальних машин*** (МВМ, VM Monitor, MVM), виконується на фізичній машині, безпосередньо взаємодіючи з її апаратним забезпеченням. Кожна ВМ – це точна копія апаратного забезпечення, на якій запускалася будь-яка ОС, реалізована для цієї архітектури. Найчастіше на ВМ встановлювалася спеціально розрахована на одного користувача ОС (підсистема діалогової обробки – ПДО). На різних ВМ могли одночасно функціонувати різні ОС.

Коли програма, написана для ПДО, виконувала системний виклик, його перехоплювала копія ПДО, запущена на відповідній ВМ. Потім ПДО виконувала відповідні апаратні інструкції, наприклад інструкції введення-виведення. Ці інструкції перехоплював МВМ і переробляв їх на апаратні інструкції фізичної машини. Віртуальні машини спільно використовували ресурси реального комп'ютера. Наприклад, дисковий простір розподілявся на віртуальні диски, що називаються ***мінідисками.***

3.9 Мережеві і розподілені ОС

Залежно від того, який віртуальний образ створює операційна система для того, щоб підмінити ним реальну апаратуру комп'ютерної мережі, відрізняють мережеві ОС і розподілені ОС. Мережева ОС надає користувачеві деяку віртуальну обчислювальну систему, працювати з якою набагато простіше, ніж з реальною мережевою апаратурою. В той же час ця віртуальна система не повністю приховує розподілену природу свого реального прототипу, тобто є віртуальною мережею.

При використанні ресурсів комп'ютерів мережі користувач мережевої ОС завжди пам'ятає, що він має справу з мережевими ресурсами. І для доступу до них треба виконати деякі особливі операції. Наприклад, відобразити віддалений каталог, що розділяється, на вигадану локальну букву дисковода або поставити перед ім'ям каталогу ще і ім'я комп'ютера, на якому той розташований. Користувачі мережевої ОС мають бути в курсі того, де зберігаються їх файли.

Працюючи в середовищі мережевої ОС, користувач хоча і може запустити завдання на будь-якій машині комп'ютерної мережі, завжди знає, на якій машині виконується його завдання. За умовчанням призначене для користувача завдання виконується на тій машині, на якій користувач зробив логічний вхід. Якщо ж він хоче виконати завдання на іншій машині, то йому треба або виконати логічний вхід в цю машину, або ввести спеціальну команду віддаленого виконання, в якій він повинен вказати інформацію, що ідентифікує віддалений комп'ютер.

Магістральним напрямом розвитку мережевих операційних систем є досягнення як можна вищогоступеня прозорості мережевих ресурсів. В ідеальному випадку мережева ОС повинна представити користувачеві мережеві ресурси у вигляді ресурсів єдиної централізованої віртуальної машини. Для такої операційної системи використовують спеціальну назву – ***розподілена ОС***, або істинно розподілена ОС.

Розподілена ОС, динамічно і автоматично розподіляючи роботи по різних машинах системи для обробки, примушує набір мережевих машин працювати як віртуальний універсальний процесор. Користувач розподіленої ОС, взагалі кажучи, не має відомостей про те, на якій машині виконується його робота.

Розподілена ОС існує як єдина операційна система в масштабах обчислювальної системи. Кожен комп'ютер мережі, працюючої під управлінням розподіленої ОС, виконує частину функцій цієї глобальної ОС. Розподілена ОС об'єднує всі комп'ютери мережі в тому сенсі, що вони працюють в тісній кооперації один з одним для ефективного використання усіх ресурсів комп'ютерної мережі.

3.10 Мультипрограмування

Мультипрограмування, або багатозадачність (multitasking), – це спосіб організації обчислювального процесу, при якому на одному процесорі чередуючись виконуються відразу декілька програм. Ці програми спільно використовують не лише процесор, але і інші ресурси комп'ютера: оперативну і зовнішню пам'ять, пристрої введення-виведення, дані. Мультипрограмування підвищує ефективність використання обчислювальної системи, проте ефективність може розумітися по-різному. Найхарактернішими критеріями ефективності обчислювальних систем є:

- пропускна спроможність – кількість завдань, що виконуються обчислювальною системою за одиницю часу;
- зручність роботи користувачів, що полягає, зокрема, в тому, що вони мають можливість інтерактивно працювати одночасно з декількома додатками на одній машині;
- реактивність системи – здатність системи витримувати заздалегідь задані інтервали часу між запуском програми і отриманням результату.

Залежно від вибраного критерію ефективності ОС діляться на системи пакетної обробки, системи розподілу часу і системи реального часу. Кожен тип ОС має специфічні внутрішні механізми і особливі сфери застосування. Деякі ОС можуть підтримувати одночасно декілька режимів.

3.10.1 Мультипрограмування в системах пакетної обробки

При використанні мультипрограмування для підвищення пропускної спроможності комп'ютера головною метою є мінімізація простоїв усіх пристроїв комп'ютера, і, перш за все, центрального процесора. Такі простої можуть виникати із-за призупинення завдання з його внутрішніх причин, пов'язаних, наприклад, з очікуванням введення даних для обробки. Дані можуть зберігатися на диску або ж поступати від користувача, працюючого за терміналом, а також від вимірювальної апаратури, встановленої на зовнішніх технічних об'єктах. При виникненні такого роду блокування виконуваного завдання природним рішенням, що веде до підвищення ефективності використання процесора, є перемикання процесора на виконання іншого завдання, в якого є дані для обробки. Така концепція мультипрограмування покладена в основу так званих пакетних систем.

Системи пакетної обробки призначалися для розв'язання задач в основному обчислювального характеру, що не вимагають швидкого отримання результатів. Головною їх метою і критерієм ефективності є максимальна пропускна спроможність, тобто розв'язання максимального числа завдань за одиницю часу.

Для досягнення цієї мети в системах пакетної обробки використовується така схема функціонування. На початку роботи формується пакет завдань, в якому кожне завдання містить вимогу до системних ресурсів. З цього пакету завдань формується мультипрограмна суміш, тобто певна кількість одночасно виконуваних завдань. Для одночасного виконання вибираються завдання, що пред'являють різні вимоги до ресурсів, так, щоб забезпечувалося збалансоване завантаження усіх пристроїв обчислювальної машини. Наприклад, в мультипрограмній суміші бажана одночасна присутність обчислювальних завдань і завдань з інтенсивним введенням-виведенням. Таким чином, вибір нового завдання з пакету завдань залежить від внутрішньої ситуації, що складається в системі, тобто вибирається «вигідне» завдання. Отже, в обчислювальних системах, працюючих під управлінням пакетних ОС, неможливо гарантувати виконання того або іншого завдання впродовж певного періоду часу.

Розглянемо детальніше поєднання в часі операцій введення-виведення і обчислень. Таке поєднання може досягатися різними способами. Один з них характерний, наприклад, для комп'ютерів, що мають спеціалізований процесор введення-виведення. У комп'ютерах класу мейнфреймів такі процесори називають ***каналами***. У системі команд центрального процесора передбачається спеціальна інструкція, за допомогою якої каналу передаються параметри і вказівки на те, яку програму введення-виведення він повинен виконати. Починаючи з цього моменту центральний процесор і канал можуть працювати паралельно (рис. 3.14, а).

Інший спосіб поєднання обчислень з операціями введення-виведення реалізується в комп'ютерах, в яких зовнішні пристрої управляються не процесором введення-виведення, а контролерами. Кожен зовнішній пристрій

(чи група зовнішніх пристроїв одного типу) має свій власний контролер, який автономно відпрацьовує команди, що поступають від центрального процесора. При цьому контролер і центральний процесор працюють асинхронно. Оскільки багато зовнішніх пристроїв включають електромеханічні вузли, контролер виконує свої команди управління пристроями істотно повільніше, ніж центральний процесор – свої.

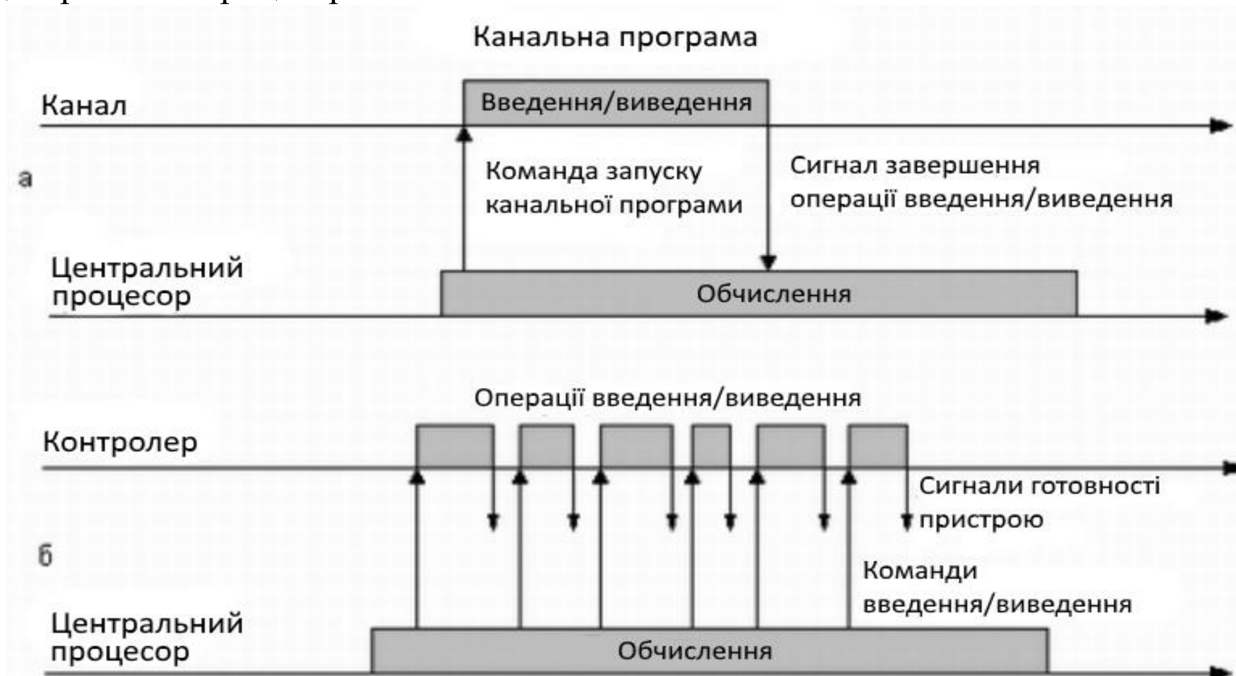


Рисунок 3.14 – Паралельне виконання обчислень і операцій введення-виведення

Ця обставина використовується для організації паралельного виконання обчислень і операцій введення-виведення: в проміжку часу між передачею команд, контролеру центральний процесор може виконувати обчислення (див. рис. 3.14, б). Контролер може повідомити центральний процесор про те, що він готовий прийняти наступну команду, сигналом переривання або центральний процесор дізнається про це, періодично опитуючи стан контролера.

Максимальний ефект прискорення досягається при найповнішому перекритті обчислень і введення-виведення. Розглянемо випадок, коли процесор виконує тільки одне завдання. У цій ситуації ступінь прискорення залежить від природи цього завдання і від того, наскільки ретельно був виявлений можливий паралелізм при її програмуванні. У завданнях, в яких переважають або обчислення, або введення-виведення, прискорення майже відсутнє. Паралелізм у рамках одного завдання неможливий також, коли для продовження обчислень потрібне повне завершення операції введення-виведення. У таких випадках неминучі простої центрального процесора або каналу.

Якщо ж в системі виконуються одночасно декілька завдань, з'являється можливість поєднання обчислень одного завдання з введенням-виведенням іншого. Поки одне завдання чекає якої-небудь події, процесор не простоює, як це відбувається при послідовному виконанні програм, а виконує інше завдання. Відмітимо, що такою подією в мультипрограмній системі може бути не лише

завершення введення-виведення, але і, наприклад, настання певного моменту часу, розблокування файлу або завантаження сторінки з диска.

Загальний час виконання суміші завдань часто виявляється меншим, ніж їх сумарний час послідовного виконання (рис. 3.15, а). Проте виконання окремого завдання в мультипрограмному режимі може зайняти більше часу, ніж при монопольному виділенні процесора цьому завданню.

Дійсно, при спільному використанні процесора в системі можуть виникати ситуації, коли завдання готове виконуватися, але процесор зайнятий виконанням іншого завдання. У таких випадках завдання, що завершило введення-виведення, готове виконуватися, але змушене чекати звільнення процесора, і це подовжує термін його виконання.

Так, з рис. 3.15 видно, що в однопрограмному режимі завдання **A** виконується за 6 одиниць часу, а в мультипрограмному – за 7. Завдання **B** також замість 5 одиниць часу виконується за 6. Та зате час виконання обох завдань в мультипрограмному режимі складає всього 8 одиниць, що на 3 одиниці менше, ніж при послідовному виконанні.

У системах пакетної обробки перемикання процесора з виконання одного завдання на виконання іншого відбувається за ініціативою найактивнішого завдання, наприклад, коли завдання відмовляється від процесора із-за необхідності виконати операцію введення-виведення. Тому існує висока ймовірність того, що одне завдання може надовго зайняти процесор і виконання інтерактивних завдань стане неможливим.

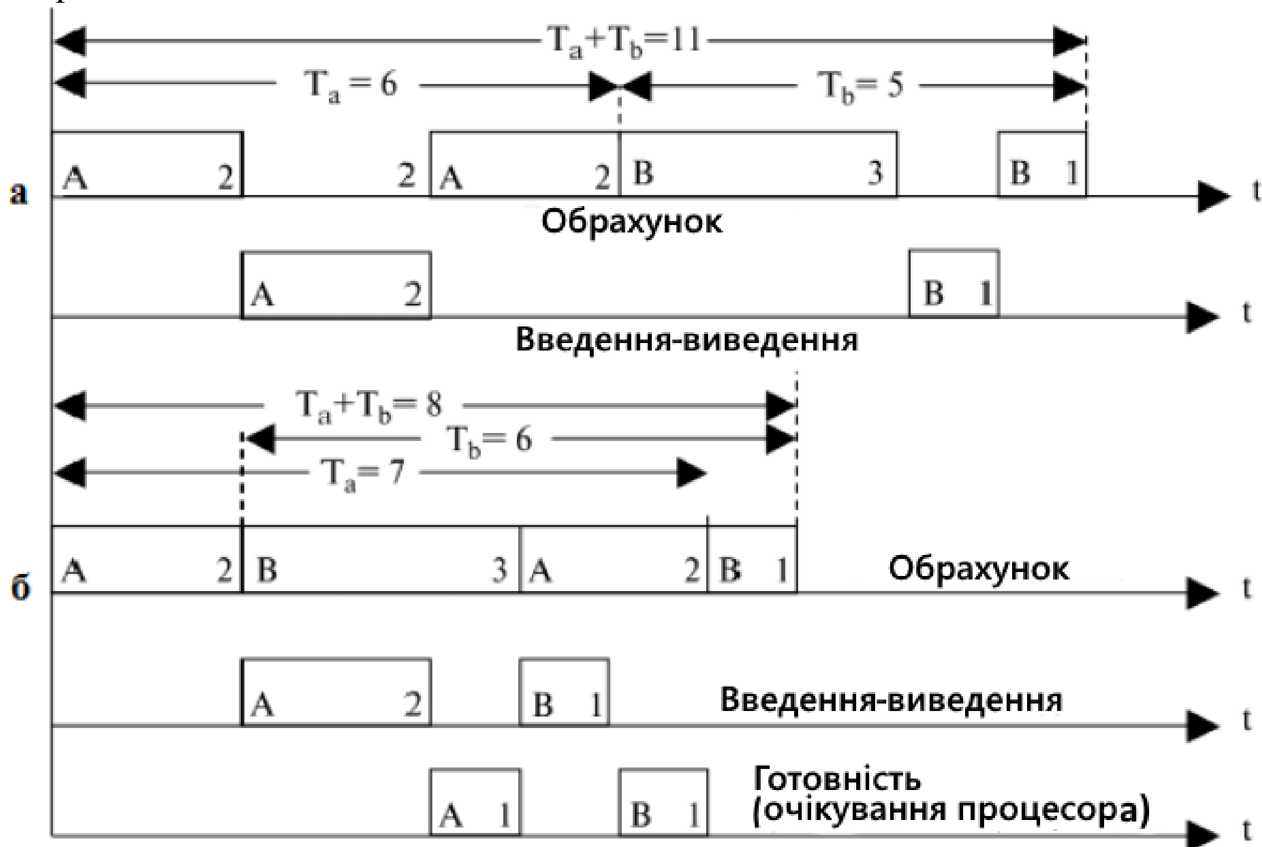


Рисунок 3.15 – Час виконання двох завдань: в однопрограмній (а), та мультипрограмній (б) системах

Взаємодія користувача з обчислювальною машиною з системою пакетної обробки зводиться до того, що він приносить завдання, віддає його диспетчерові-операторові, а в кінці дня після виконання усього пакету завдань отримує результат. Очевидно, що такий порядок підвищує ефективність функціонування апаратури, але знижує ефективність роботи користувача.

3.10.2 Мультипрограмування в системах розподілу часу

Підвищення зручності і ефективності роботи користувача є метою іншого способу мультипрограмування – розподілу часу. У системах розподілу часу користувачам (чи одному користувачеві) надається можливість інтерактивної роботи відразу з декількома додатками. Для цього кожен додаток повинен регулярно одержувати можливість «спілкування» з користувачем. Зрозуміло, що в пакетних системах можливості діалогу користувача з додатком дуже обмежені.

У системах розподілу часу ця проблема вирішується за рахунок того, що ОС примусово періодично призупиняє додатки, не чекаючи, коли вони добровільно звільнять процесор. Усім додаткам поперемінно виділяється квант процесорного часу, таким чином користувачі, що запустили програми на виконання, одержують можливість підтримувати з ними діалог.

Системи розподілу часу покликані виправити основний недолік систем пакетної обробки – ізоляцію користувача-програміста від процесу виконання його завдань. Кожному користувачеві в цьому випадку надається термінал, з якого він може вести діалог зі своєю програмою (рис. 3.16).

Рисунок 3.16 – Система розподілу часу

Оскільки в системах розподілу часу кожному завданню виділяється тільки квант процесорного часу, жодне завдання не займає процесор надовго і час відповіді виявляється прийнятним. Якщо квант вибраний досить невеликим, то в усіх користувачів, одночасно працюючих на одній і тій же машині, складається враження, що кожен з них одноосібно використовує машину.

Ясно, що системи розподілу часу мають меншу пропускну спроможність, чим системи пакетної обробки, оскільки на виконання приймається кожне запущене користувачем завдання, а не те, яке «вигідне» системі.

Крім того, продуктивність системи знижується із-за збільшених накладних витрат обчислювальної потужності на частіше перемикання процесора із завдання на завдання. Це цілком відповідає тому, що критерієм ефективності

систем розподілу часу є не максимальна пропускна спроможність, а зручність і ефективність роботи користувача. В той же час мультипрограмне виконання інтерактивних додатків підвищує і пропускну спроможність комп'ютера (хай і не в такому ступені, як пакетні системи). Апаратура завантажується краще, оскільки в той час, поки один додаток чекає повідомлення користувача, інші додатки можуть оброблятися процесором.

3.10.3 Мультипрограмування в системах реального часу

Ще один різновид мультипрограмування використовується в системах реального часу, призначених для управління від комп'ютера різними технічними об'єктами (наприклад, верстатом, супутником, науковою експериментальною установкою тощо) або технологічними процесами (наприклад, гальванічною лінією, доменним процесом тощо).

В усіх цих випадках існує гранично допустимий час, впродовж якого має бути виконана та або інша програма, що управляє об'єктом. Інакше може статися аварія: супутник вийде із зони видимості, експериментальні дані, що поступають з датчиків, будуть втрачені, товщина гальванічного покриття не відповідатиме нормі.

Таким чином, критерієм ефективності тут є здатність витримувати заздалегідь задані інтервали часу між запуском програми і отриманням результату (дії, що управляє). Цей час називається часом реакції системи, а відповідна властивість системи – реактивністю. Вимоги до часу реакції залежать від специфіки керованого процесу. Контролер робота може вимагати від вбудованого комп'ютера відповідь протягом 1 мс, тоді як при моделюванні польоту може бути прийнятна відповідь в 40 мс.

У системах реального часу мультипрограмна суміш є фіксованим набором заздалегідь розроблених програм, а вибір програми на виконання здійснюється за перериваннями (виходячи з поточного стану об'єкту) або відповідно до розкладу планових робіт.

Здатність апаратури комп'ютера і ОС до швидкої відповіді залежить в основному від швидкості перемикання з одного завдання на інше і, зокрема, від швидкості обробки сигналів переривання. Якщо при виникненні переривання процесор повинен опитати сотні потенційних джерел переривання, то реакція системи буде занадто повільною. Час обробки переривання в системах реального часу часто визначає вимоги до класу процесора навіть при невеликому його завантаженні.

У системах реального часу не прагнуть максимально завантажувати всі пристрої, навпаки, при проектуванні програмного комплексу, що управляє, закладається деякий «запас» обчислювальної потужності на випадок пікового навантаження. Статистичні аргументи про низьку ймовірність виникнення пікового навантаження, грунтуються на тому, що ймовірність одночасного виникнення великої кількості незалежних подій дуже мала, і не застосована до багатьох ситуацій в системах управління.

3.11 Багатопоточність

Однією з основних концепцій, що допомагають зрозуміти структуру ОС, є концепція процесів. Є багато визначень терміну «процес», у тому числі:

- програма, що виконується;
- об'єкт, який можна ідентифікувати і виконати на комп'ютері;
- одиниця активності, яку можна охарактеризувати поточним станом і пов'язаним з нею набором ресурсів.

Процес можна розділити на чотири компоненти:

- програма, що виконується;
- дані і ресурси, потрібні для її роботи;
- контекст виконання програми.

Багатопоточність (multithreading) – це технологія, при якій процес, що виконується додатком, розділяється на декілька одночасно виконуваних ***потоків***. Багатопоточність виявляється дуже корисною для додатків, що виконують декілька незалежних завдань, які не вимагають послідовного виконання. Якщо в межах одного і того ж процесу обробляється декілька потоків, то при перемиканні між різними потоками непродуктивна витрата ресурсів процесора менша, ніж при перемиканні між різними процесами.

3.12 Симетрична багатопроцесорна обробка

До недавнього часу усі персональні комп'ютери, розраховані на одного користувача, і робочі станції містили один процесор загального призначення. В результаті постійного підвищення вимог до продуктивності і пониження вартості процесорів виробники перейшли до випуску комп'ютерів з декількома процесорами. Для підвищення ефективності і надійності використовується технологія ***симетричної багатопроцесорності*** (Symmetric MultiProcessing – SMP). Цей термін належить до архітектури апаратного забезпечення комп'ютера, а також до образу дій ОС, що відповідає цій архітектурній особливості. Симетричну багатопроцесорність можна визначити, як автономну комп'ютерну систему з такими характеристиками:

1. У системі є декілька процесорів.
2. Ці процесори, сполучені між собою комунікаційною шиною, спільно використовують одну і ту ж основну пам'ять і одні і ті ж пристрої введення-виведення.
3. Усі процесори можуть виконувати одні і ті ж функції (звідси назва ***симетрична обробка***).

ОС, працююча в системі з симетричною багатопроцесорністю, розподіляє процеси або потоки між усіма процесорами. У багатопроцесорних систем є декілька потенційних переваг в порівнянні з однопроцесорними.

Продуктивність. Якщо завдання можна організувати так, що його певні частини виконуватимуться паралельно, це призведе до підвищення продуктивності порівняно з однопроцесорними системами.

Надійність. При симетричній мультипроцесорній обробці відмова одного процесора не призведе до зупинки машини.

Нарощування. Додаючи в систему додаткові процесори, можна підвищити продуктивність системи.

Масштабованість. Виробники можуть пропонувати свої продукти в різних конфігураціях, що відрізняються ціною і продуктивністю.

3.12.1 Архітектура симетричної багатопроцесорності

Найранішою і найвідомішою є класифікація архітектури обчислювальних систем, запропонована в 1966 році М. Флінном. Класифікація базується на понятті ***потоку***, під яким розуміється послідовність елементів, команд або даних, що обробляється процесором. На основі числа потоків команд і потоків даних Флінн виділяє чотири класи архітектури: **SISD, MISD, SIMD, MIMD** (ми розглянемо два). Ці чотири класи архітектури схематично представляються у вигляді квадрата, що називається квадратом Флінна (рис. 3.17).

		Data Stream	
		Single	Multiple
Instruction Stream	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Рисунок 3.17 – Класифікація Флінна

На рисунках, що ілюструють класифікацію М. Флінна, використані такі позначення:

- ПР – один або декілька процесорних елементів;
- ПУ – пристрій управління;
- ПД – пам'ять даних.

SISD (Single Instruction stream / Single Data stream) – одиночний потік команд і одиночний потік даних. До цього класу належать класичні послідовні машини, або інакше, машини фон-нейманівського типу, наприклад, PDP-11, IBM PC (рис. 3.18). У таких машинах один управляючий пристрій, він отправляє тільки один потік команд, усі команди обробляються послідовно одна за одною одним процесором, і кожна команда ініціює одну операцію з одним потоком даних. Не має значення той факт, що для збільшення швидкості обробки команд і швидкості виконання арифметичних операцій може застосовуватися конвеєрна обробка – як машина CDC 6600 із скалярними функціональними пристроями, так і CDC 7600 з конвеєрними пристроями потрапляють в цей клас.

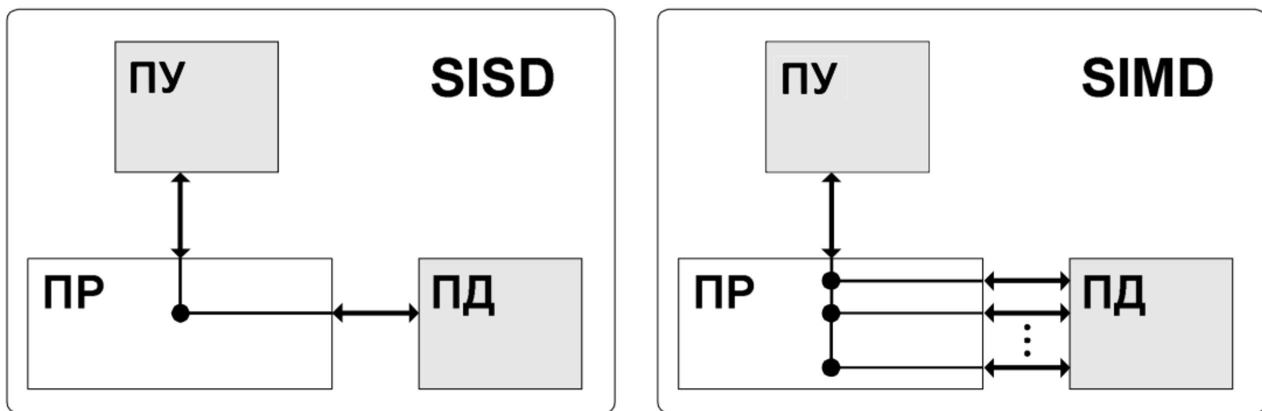


Рисунок 3.18 – Архітектура SISD і SIMD

SIMD (Single Instruction stream / Multiple Data stream) – поодинокий потік команд і множинний потік даних. Ці системи мають один управляючий пристрій і велику кількість процесорів, які можуть виконувати одну і ту ж інструкцію щодо різних даних. Єдина інструкція паралельно виконується над багатьма елементами даних. Архітектура подібного роду включає, на відміну від попереднього класу, векторні команди. Це дозволяє виконувати одну арифметичну операцію відразу над багатьма даними – елементами вектору. Спосіб виконання векторних операцій не обмовляється, тому обробка елементів вектора може здійснюватися або процесорною матрицею, як в ILLIAC IV, або за допомогою конвеєра, як, наприклад, в машині CRAY-1 (див. рис. 3.18).

Машини типу SIMD складаються з великого числа ідентичних процесорних елементів, що мають власну пам'ять. Усі процесорні елементи в такій машині виконують одну і ту ж програму. Очевидно, що така машина, складена з великого числа процесорів, може забезпечити дуже високу продуктивність тільки на тих задачах, при розв'язанні яких усі процесори можуть робити одну і ту ж роботу. Модель обчислень для машини SIMD дуже схожа на модель обчислень для векторного процесора: поодинока операція виконується над великим блоком даних. На відміну від обмеженого конвеєрного функціонування векторного процесора, матричний процесор (синонім для більшості SIMD-машин) може бути значно гнучкішим. Оброблювальні елементи таких процесорів – це універсальні програмовані ЕОМ, так що задачі, що розв'язуються паралельно, можуть бути досить складними і містити галуження.

Моделі обчислень на векторних і матричних ЕОМ настільки схожі, що ці ЕОМ часто вважаються як еквівалентними.

Майже усі комп'ютери сьогодні реалізують певну форму набору команд SIMD. Процесори Intel реалізують набори команд MMX, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2) і Streaming SIMD Extensions 3 (SSE3), які можуть обробляти декілька елементів даних за один такт.

Безперечними представниками класу SIMD вважаються матриці процесорів: ILLIAC IV, ICL DAP, Goodyear Aerospace MPP тощо. У таких системах єдиний управляючий пристрій контролює певну кількість процесорних елементів. Кожен процесорний елемент отримує від пристроїв управління в

кожен фіксований момент часу однакову команду і виконує її над своїми локальними даними. Для класичних процесорних матриць ніяких питань не виникає, проте в цей же клас можна включити і векторно-конвеєрні машини, наприклад, CRAY-1.

MISD (Multiple Iinstruction stream / Single Data stream) – множинний потік команд і поодинокий потік даних. Це архітектура багатьох процесорів, які оброблюють один і той же потік даних (рис. 3.19). Проте ні Флінн, ні інші фахівці в області архітектури комп'ютерів досі не змогли представити приклад реально існуючої обчислювальної системи, побудованої на цьому принципі. В більшості випадків декільком потокам команд потрібні декілька потоків даних, так що цей клас паралельних комп'ютерів застосовується дослідниками лише як теоретична модель, а не як реальний комп'ютер масового виробництва. Вважається, що дотепер цей клас порожній. Але не варто вважати це недоліком схеми. Така архітектура в майбутньому може стати надзвичайно корисною для розробки принципово нових обчислювальних систем.

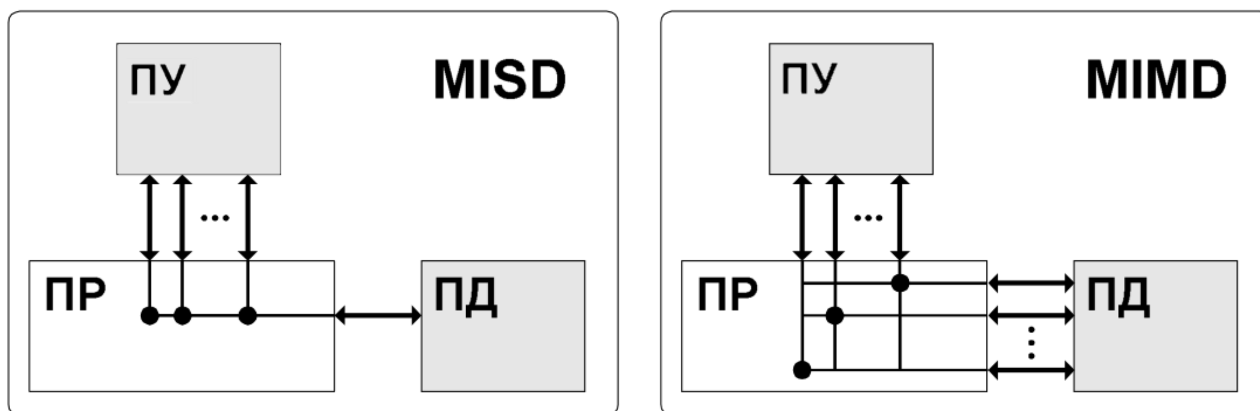


Рисунок 3.19 – Архітектура MISD і MIMD

MIMD (multiple instruction stream / multiple data stream) – множинний потік команд і множинний потік даних. У цій архітектурі кожен процесор має свій управляючий пристрій і незалежно виконує різні набори команд, що обробляють різні набори даних (див. рис. 3.19). Системи в архітектурі MIMD діляться на системи з розподіленою пам'яттю (***слабкозв'язані системи***), до яких належать кластери, і системи з спільно використовуваною пам'яттю. До останніх належать симетричні мультипроцесорні системи. В наші дні це – найпопулярніший тип паралельного комп'ютера. Нові багатоядерні платформи (такі, як процесор Intel Core Duo) потрапляють саме в цю категорію.

Термін «мультипроцесор» покриває більшість машин типу MIMD і (подібно до того, як термін «матричний процесор» застосовується до машин типу SIMD) часто використовується як синонім для машин типу MIMD. У мультипроцесорній системі кожен процесорний елемент виконує свою програму досить незалежно від інших процесорних елементів. Процесорні елементи, звичайно, повинні якось зв'язуватися один з одним, що робить необхідним детальнішу класифікацію машин типу MIMD.

У мультипроцесорах із загальною пам'яттю (***сильнозв'язанних мультипроцесорах***) є пам'ять даних і команд, доступна усім процесорним

елементам. Із загальною пам'яттю процесорні елементи зв'язуються за допомогою загальної шини або мережі обміну. В протилежність цьому варіанту в слабкозв'язаних багатопроцесорних системах (машинах з локальною пам'яттю) уся пам'ять ділиться між процесорними елементами і кожен блок пам'яті доступний тільки пов'язаному з ним процесору. Мережа обміну зв'язує процесорні елементи один з одним.

Базовою моделлю обчислень на MIMD-мультипроцесорі є сукупність незалежних процесів, які епізодично звертаються до даних, що розділяються. Існує велика кількість варіантів цієї моделі. На одному кінці спектру – модель розподілених обчислень, в якій програма ділиться на досить велике число паралельних завдань, що складаються з декількох підпрограм. На іншому кінці спектру – модель потокових обчислень, в яких кожна операція в програмі може розглядатися як окремий процес. Така операція чекає своїх вхідних даних (операндів), які мають бути передані їй іншими процесами. Після їх отримання операція виконується, і отримане значення передається тим процесам, які його потребують.

Клас MIMD надзвичайно широкий, оскільки включає всілякі мультипроцесорні системи: CRAY Y-MP, Intel Paragon, CRAY T3D і багато інших. Цікаве те, що якщо конвеєрну обробку розглядати як виконання декількох команд не над поодиноким векторним потоком даних, а над множинним скалярним потоком, то усі розглянуті вище векторно-конвеєрні комп'ютери можна розташувати і в цьому класі. Подальша класифікація обчислювальних систем з архітектурою MIMD може виконуватися відповідно до того, як в них здійснюється обмін даними між процесорами.

У системі MIMD процесори є універсальними, тому що вони повинні мати можливість обробляти усі команди, необхідні для відповідного перетворення даних. Якщо кожному процесору виділяється окрема ділянка пам'яті, то кожен такий елемент є самостійним комп'ютером. Вони обмінюються між собою інформацією або через спеціальні канали, або через деякі мережеві пристрої. Такі системи відомі як ***кластери***, або ***мультикомп'ютери***. Якщо процесори спільно використовують загальну пам'ять, то кожен з них має доступ до програм і даних, які там зберігаються. Такі системи відомі під назвою ***багатопроцесорних систем із загальною пам'яттю***.

Одна з класифікацій багатопроцесорних систем грунтується на тому, як процеси розподіляються між процесорами. Існують два головні підходи – виділення основних і підпорядкованих процесорів і симетрична багатопроцесорна обробка. В архітектурі з ***ведучим*** і ***веденими*** процесорами ядро ОС завжди виконується на спеціально виділеному процесорі. На інших процесорах можуть виконуватися тільки програми користувача і, можливо, утиліти ОС.

Провідний процесор відповідає за планування процесів або потоків. Якщо процесу, що виконується на веденому процесорі, знадобиться який-небудь системний сервіс, він повинен послати запит основному процесору і потім чекати, поки сервісна програма не закінчить свою роботу. Для реалізації такого підходу досить удосконалити ОС, призначену для однопроцесорних

багатозадачних систем. Вирішення конфліктів спрощується, завдяки тому, що усією пам'яттю і усіма ресурсами введення-виведення управляє один процесор. Цей підхід має ряд недоліків:

1. Збій в роботі основного процесора призводить до відмови усієї системи.
2. Основний процесор може гальмувати роботу усієї системи, оскільки тільки на ньому повинні виконуватися усі дії з планування і управління процесами.

У симетричній багатопроцесорній системі ядро ОС може виконуватися на будь-якому процесорі. Як правило, кожен процесор сам планує свою роботу. Ядро може бути виконане у вигляді багатьох процесів або багатьох потоків, при цьому різні його частини здатні працювати паралельно. Симетричний підхід дещо ускладнює архітектуру ОС. Потрібно вжити запобіжні заходи, щоб два процесори не вибрали один і той же процес, або щоб процес яким-небудь чином не випав з черги. Необхідно застосувати спеціальні методи для дозволу запитів одного і того ж ресурсу різними процесами і синхронізації запитів.

3.12.2 Організація симетричної багатопроцесорної системи

Архітектура SMP-системи має декілька процесорів, кожен з яких містить свій власний управляючий модуль, арифметично-логічний пристрій і свої регістри. Кожен з процесорів має доступ до загальної основної пам'яті і до пристроїв уведення-виведення. Цей доступ здійснюється за допомогою деякого механізму взаємодії. Традиційно в такій ролі виступає загальна шина. Процесори можуть обмінюватися між собою інформацією через загальну пам'ять. Крім того, процесори можуть мати можливість безпосереднього обміну сигналами.

Як правило, в сучасних машинах процесори мають, принаймні, один рівень власного кеша. Це вносить деякі нюанси в архітектуру ОС. Оскільки в кожному локальному кеші зберігається образ якоїсь частини основної пам'яті, то в результаті зміни слова в одному кеші відповідне слово в іншому кеші може виявитися невірним. Для запобігання цьому усі процесори, кеш яких містить це слово, мають бути сповіщені про необхідність змінити його. Ця проблема відома як ***когерентності кешів*** (від лат. cohaerens – що знаходиться в зв'язку) і вирішується на апаратному рівні.

3.12.3 Архітектура багатопроцесорних ОС

ОС, призначена для симетричної багатопроцесорної системи, управляє процесорами і іншими ресурсами комп'ютера так, щоб з точки зору користувача багатопроцесорна система виглядала так само, як і багатозадачна однопроцесорна. До числа особливостей архітектури багатопроцесорних ОС входять такі.

Одночасні паралельні процеси або потоки. Щоб декілька процесів могли одночасно виконувати один і той же код ядра, він має бути реєнтерабельним. При виконанні декількома процесорами одного і того ж коду ядра (чи різних його частин) потрібна організація управління таблицями і структурами ядра, щоб уникнути взаємоблокувань або неправильного виконання операції.

Планування. Планування може виконуватися на будь-якому з процесорів, тому необхідно передбачити механізм, що дозволяє уникнути конфліктів. При використанні багатопоточності на рівні ядра декілька потоків одного і того ж процесу можуть виконуватися на різних процесорах. Планування в багатопроцесорних системах розглядатиметься далі.

Синхронізація, синхронна і асинхронна взаємодія. За ситуації, коли декілька активних процесів мають можливість доступу до спільних адресних просторів або ресурсів введення-виведення, необхідного потурбуватись про їх ефективну синхронізацію.

Синхронізація – це засіб, який забезпечує реалізацію взаємовиключень і впорядкування подій. Загальноприйнятим механізмом синхронізації в багатопроцесорних ОС є блокування.

При описі взаємодії між елементами програмних систем ініціатор взаємодії, тобто компонент, що посилає запит на обробку, називається ***клієнтом***, а компонент, що обробляє запит – ***сервером.*** У більшості випадків один і той же компонент може виступати в різних ролях – то клієнта, то сервера – в різних взаємодіях. Лише в невеликому класі систем ролі клієнта і сервера закріплюються за компонентами на увесь час їх існування.

Синхронною (блокуючою) називається така взаємодія між компонентами, при якій клієнт, відіславши запит, блокується і може продовжувати роботу тільки після отримання відповіді від сервера (рис. 3.20).

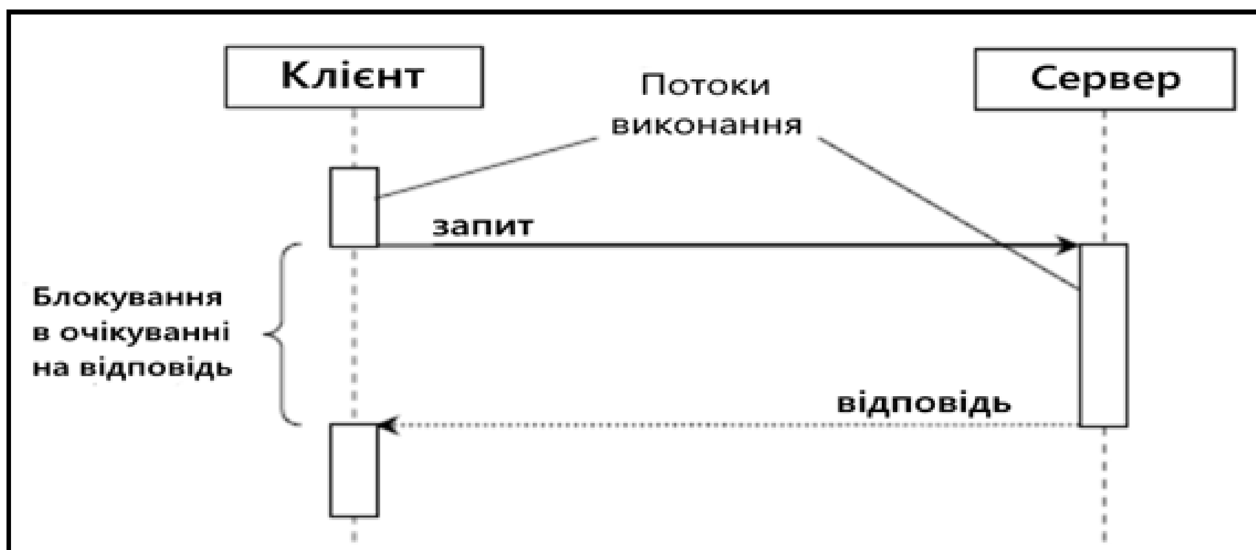


Рисунок 3.20 – Синхронна взаємодія

Синхронну взаємодію досить просто організувати, і вона набагато простіша для розуміння. Людині простіше розуміти процеси, які розгортаються послідовно, оскільки не треба постійно перемикати увагу на різні події, що відбуваються одночасно. Код програми клієнтського компонента, що описує синхронну взаємодію, влаштований простіше.

В той же час синхронна взаємодія веде до значних витрат часу на очікування відповіді. Цей час часто можна використати кориснішим чином – чекаючи відповіді на один запит, клієнт міг би зайнятися іншою роботою.

У рамках ***асинхронної*** або ***неблокуючої*** взаємодії клієнт після відправки запиту серверу може продовжувати роботу, навіть якщо відповідь на запит ще не прийшла. Асинхронна взаємодія дозволяє отримати вищу продуктивність системи за рахунок використання часу між відправкою запиту і отриманням відповіді на нього для виконання інших задач (рис. 3.21). Інша важлива перевага асинхронної взаємодії – менша залежність клієнта від сервера, можливість продовжувати роботу, навіть якщо машина, на якій знаходиться сервер, стала недоступною.

В той же час асинхронну взаємодію набагато складніше організувати, розробляти і супроводжувати. Оскільки при такій взаємодії треба писати специфічний код для отримання і обробки результатів запитів.

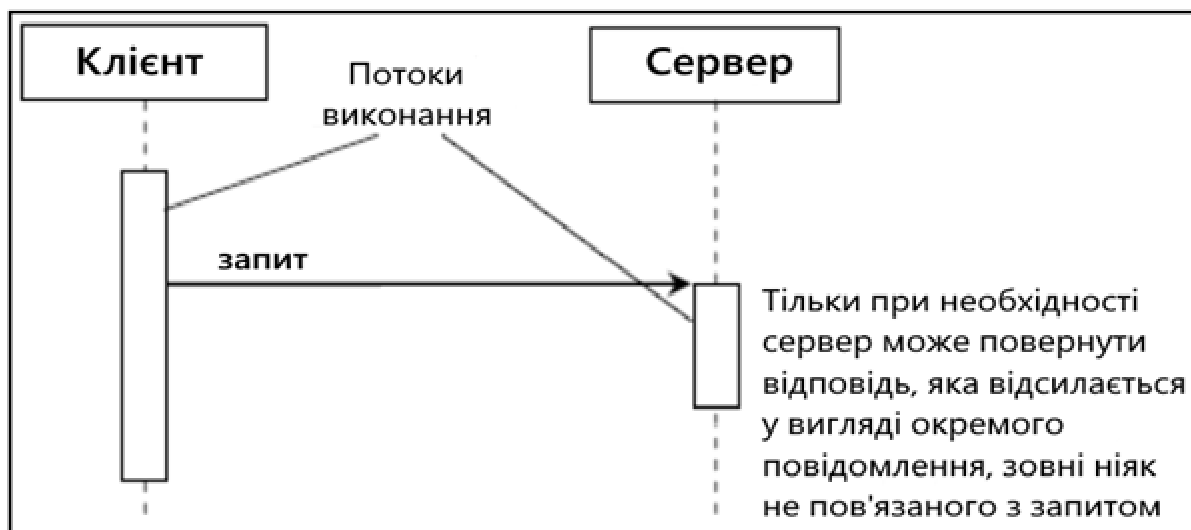


Рисунок 3.21 – Асинхронна взаємодія

Управління пам'яттю. Система управління пам'яттю в багатопроцесорній системі має бути здатна вирішувати усі проблеми, які виникають в однопроцесорних комп'ютерах. Крім того, ОС повинна уміти використати можливості, що надаються апаратним забезпеченням. Механізми сторінкової організації пам'яті різних процесорів мають бути скоординовани, щоб забезпечити узгодженість дій за ситуації, коли декілька процесорів використовують одну і ту ж сторінку або один і той же сегмент.

Надійність і відмовостійкість. При відмові одного з процесорів ОС повинна забезпечити продовження роботи системи. Планувальник ОС (як і інші його частини) повинен отримати інформацію про втрату одного з процесорів і відповідним чином перебудувати свої управляючі таблиці.

Контрольні питання і тести до розділу 3

Контрольні питання

1. Які експлуатаційні і ринкові вимоги, окрім вимог функціональної повноти, пред'являються до операційних систем?
2. За рахунок чого досягається розширюваність ОС?
3. Написання переносимої ОС аналогічно написанню будь-якого переносимого коду. Яким правилам необхідно слідувати для написання переносимої ОС?
4. На яких рівнях досягається сумісність ОС?
5. У рамках якого проекту ведуться роботи по стандартизації інтерфейсу ОС?
6. Які вбудовані інструменти загального призначення підтримують різні механізми захисту і безпеку ОС?
7. Відповідно до вимог Помаранчевої Книги, до якого рівня безпеки належать ОС?
8. На які групи розділяються модулі при загальному підході до структуризації операційної системи?
9. Які функції виконуються модулями ядра?
10. На які групи підрозділяються допоміжні модулі ОС?
11. Як завантажуються в оперативну пам'ять модулі ОС, оформлені у вигляді утиліт, системних оброблювальних програм і бібліотек?
12. Які режими роботи підтримує апаратура комп'ютера?
13. Назвіть синоніми терміну «привілейований режим»?
14. Які популярні ОС використовують архітектуру ОС, засновану на привілейованому ядрі і додатках режиму користувача?
15. Назвіть відмінні риси більшості сучасних ОС, працюючих на монолітному ядрі.
16. Чи можуть бути монолітні системи трохи структурованими?
17. Що собою являють багаторівневі (багатошарові) системи?
18. Де і коли була розроблена перша багатошарова операційна ситема? Скільки шарів (рівнів) було в цій системі?
19. Які засоби апаратної підтримки ОС беруть участь в організації обчислювальних процесів?
20. Які сервіси підтримує модель ОС типу клієнт-сервер?
21. Які модулі зазвичай входять до складу мікроядра?
22. У чому суть мікроядерної архітектури ОС?
23. Скільки відбувається перемикань режимів роботи при виконанні системного виклику при класичній організації ОС і при мікроядерній організації?
24. Яким вимогам задовольняють сучасні операційні системи, побудовані з використанням моделі клієнт-сервер і концепції мікроядра?
25. Який головний недолік мікроядерного підходу архітектури ОС?
26. Які види ядер застосовуються в сучасних операційних системах?
27. Який підхід, прийнятий на озброєння багатьма відомими фірмами, є однією з найперспективніших в конструюванні сучасних ОС?

Тести

1. Мультитермінальний режим роботи припускає поєднання:
   1) діалогового режиму роботи і режиму мультипрограмування;
   2) привілейованого режиму роботи і режиму користувача;
   3) багатопроцесорного режиму роботи і режиму введення-виведення.
2. Ядро операційної системи на сучасних процесорах функціонує:
   1) у режимі реального часу;
   2) у режимі користувача;
   3) у привілейованому режимі;
   4) в інтерактивному режимі.
3. Мікроядерна архітектура в порівнянні з монолітним ядром має такий недолік:
   1) менш продуктивна;
   2) менш гнучка;
   3) менш відмовостійка;
   4) вимогливіша до ресурсів.
4. Резидентними називаються ті модулі ядра операційної системи, які:
   1) завантажуються в оперативну пам'ять тільки на час виконання своїх функцій;
   2) постійно знаходяться в оперативній пам'яті;
   3) зберігаються в зовнішній пам'яті.
5. Класична архітектура операційної системи припускає, що в привілейованому режимі працюють:
   1) утиліти ОС;
   2) системні управлячі програми;
   3) стандартні додатки;
   4) ядро ОС.
6. Розширюваність в ОС на основі мікроядра (в порівнянні з класичною архітектурою) досягається:
   1) складніше;
   2) рідше;
   3) так само;
   4) легше.
7. У якому режимі працюють додатки?
   1) режимі супервізора;
   2) режимі ядра;
   3) режимі користувача;
   4) привілейованому режимі.
8. До переваг мікроядерної архітектури можна віднести наступне:
   1) розширюваність, продуктивність;
   2) розширюваність, продуктивність, надійність;
   3) розширюваність, надійність, переносимість;
   4) продуктивність, надійність.
9. У ОС на основі мікроядра при зверненні до функції ОС, оформленої у вигляді сервера, зміна режимів відбувається … рази.

1) 2;
2) 3;
3) 4;
4) 5.

10. Якщо код ОС написаний так, що доповнення і зміни можуть вноситися без порушення цілісності системи, то таку ОС називають:
    1) структуризованою;
    2) розширюваною;
    3) оновлюваною;
    4) переносимою.
11. Високу міру переносимості мають ОС, побудовані відповідно до концепції:
    1) мікроядерної архітектури;
    2) класичної архітектури;
    3) мікропроцесорної архітектури.
12. Для забезпечення високої швидкості роботи ОС в оперативній пам'яті повинні розташовуватися:
    1) модулі ядра;
    2) модулі ядра і усі допоміжні модулі;
    3) допоміжні модулі;
    4) спеціальні модулі.
13. Багатошарова організація ОС істотно … розробку і модернізацію системи.
    1) ускладнює;
    2) спрощує;
    3) не впливає на.
14. Процес в мультипрограмному режимі може виконуватися швидше, ніж в монопольному:
    1) так;
    2) ні.
15. Менеджери ресурсів при мікроядерній архітектурі працюють в:
    1) режимі ядра;
    2) захищеному режимі;
    3) привілейованому режимі;
    4) режимі користувача.
16. При класичній архітектурі ядра системний виклик супроводжується:
    1) чотирма перемиканнями режиму (привілейований/користувача);
    2) двома перемиканнями режиму (привілейований/користувача);
    3) трьома перемиканнями режиму (привілейований/користувача);
    4) одним перемиканням режиму (привілейований/користувача).
17. При микроядерній архітектурі системний виклик супроводжується:
    1) одним перемиканням режиму (привілейований/користувача);
    2) двома перемиканнями режиму (привілейований/користувача);
    3) трьома перемиканнями режиму (привілейований/користувача);
    4) чотирма перемиканнями режиму (привілейований/користувача).

18. Які властивості ОС відповідають за реалізацію такої характеристики: користувачі не можуть отримати доступ до інформації і послуг без відповідного дозволу?
    1) захищеність;
    2) мобільність;
    3) масштабованість;
    4) розширюваність.
19. Які властивості ОС відповідають за реалізацію такої характеристики: ОС функціонує на різних конфігураціях апаратних засобів?
    1) захищеність;
    2) мобільність;
    3) масштабованість;
    4) розширюваність.
20. Які властивості ОС відповідають за реалізацію такої характеристики: ОС підтримує пристрої, які недоступні в період її розробки?
    1) захищеність;
    2) мобільність;
    3) масштабованість;
    4) розширюваність.
21. Які властивості ОС відповідають за реалізацію наступної характеристики: продуктивність системи стабільно росте при установці додаткової пам'яті і процесорів?
    1) захищеність;
    2) мобільність;
    3) масштабованість;
    4) розширюваність.
22. Чому перехід до використання мікроядрової архітектури (модель клієнт-сервер) може спричинити зниження продуктивності ОС? Тому що:
    1) рішення про пріоритети потоків і дисципліну постановки в чергу процесів приймає працюючий поза ядром Планувальник;
    2) при розміщенні драйверів пристроїв поза мікроядром для забезпечення можливості дозволу і заборони переривань, частина програми драйвера повинна виконуватися тільки в просторі ядра;
    3) системний виклик виконується повільніше, ніж виклик функції, реалізованої в режимі користувача, оскільки процесор двічі перемикається між режимами;
    4) замість двох перемикань режиму процесора в разі системного виклику здійснюється чотири (два – під час обміну між клієнтом і мікроядром, два – між сервером і мікроядром).
23. Зовнішні по відношенню до мікроядра компоненти ОС взаємодіють за допомогою:
    1) спеціальних каналів;
    2) механізму виклику віддалених процедур;
    3) відображення файлів в оперативну пам'ять;
    4) обміну повідомленнями, що передаються через мікроядро.

4 АПАРАТНА ПІДТРИМКА РОБОТИ ОС

Операційна система тісно пов'язана з устаткуванням комп'ютера, на якому вона повинна працювати. ОС обслуговує користувачів, звертаючись при цьому до ресурсів апаратного забезпечення, до складу яких входить один або декілька процесорів. Окрім цього, вона управляє вторинною пам'яттю і пристроями введення-виведення. Апаратне забезпечення впливає на набір команд ОС і управління його ресурсами.

Значна частина операційних систем успішно працюють на різних апаратних платформах без істотних змін у своєму складі. Це пояснюється тим, що, незважаючи на відмінності в деталях, засоби апаратної підтримки ОС більшості комп'ютерів набули сьогодні багато типових рис. У результаті серед ОС можна виділити прошарок машинно-залежних компонентів ядра і зробити інші шари ОС загальними для різних апаратних платформ. Це повною мірою стосується і популярного сімейства 32-розрядних процесорів Intel: 80386, 80486, Pentium, Pentium Pro, Pentium II, Celeron і Pentium III-IV. Слід зазначити, що засоби підтримки операційної системи в усіх цих процесорах побудовані майже ідентично, тому далі у тексті для їх позначення ми будемо використовувати узагальнений термін «процесори Pentium».

4.1 Типові засоби апаратної підтримки ОС

Чіткої межі між програмною і апаратною реалізацією функцій ОС не існує – рішення про те, які функції ОС виконуватимуться програмно, а які апаратно, приймається розробниками апаратного і програмного забезпечення комп'ютера. Проте, практично усі сучасні апаратні платформи мають деякий типовий набір засобів апаратної підтримки ОС, в який входять такі компоненти:

- засоби підтримки привілейованого режиму;
- засоби трансляції адрес;
- засоби перемикання процесів;
- система переривань і системний таймер;
- засоби захисту областей пам'яті.

4.1.1 Засоби підтримки привілейованого режиму

Засоби підтримки ***привілейованого*** (***захищеного***) ***режиму*** грунтуються на системному регістрі процесора, що часто називається «словом стану» машини або процесора. Цей регістр містить деякі ознаки, що визначають режими роботи процесора, у тому числі і ознаку поточного режиму привілеїв.

Зміна режиму привілеїв виконується за рахунок зміни слова стану машини в результаті переривання або виконання привілейованої команди. Число градацій привілейованості може бути різним у різних типів процесорів, найчастіше використовуються два рівні (ядро-користувач) або чотири (наприклад, ядро-супервізор-виконання-користувач у платформи VAX або 0-1-2-3 у процесорів Intel x86/Pentium). В обов'язки засобів підтримки

привілейованого режиму входить перевірка допустимості виконання активною програмою інструкцій процесора при поточному рівні привілейованості.

Основним режимом роботи процесора Pentium є захищений режим (protected mode). Для сумісності з програмним забезпеченням, розробленим для попередніх моделей процесорів Intel (головним чином, моделі 8086), в процесорах Pentium передбачений так званий ***реальний режим*** (real mode). У реальному режимі процесор Pentium виконує 16-розрядні інструкції і адресує один мегабайт пам'яті.

4.1.2 Засоби трансляції адрес

Засоби трансляції адрес виконують операції перетворення віртуальних адрес, які містяться в кодах процесу, в адреси фізичної пам'яті. Таблиці, призначені при трансляції адрес, мають великий об'єм, тому для їх зберігання використовуються області оперативної пам'яті, а апаратура процесора містить тільки покажчики на ці області. Засоби трансляції адрес використовують ці покажчики для доступу до елементів таблиць і апаратного виконання алгоритму перетворення адреси, що значно прискорює процедуру трансляції в порівнянні з її чисто програмною реалізацією.

4.1.3 Засоби перемикання процесів

Засоби перемикання процесів призначені для швидкого збереження контексту призупиненого процесу, і відновлення контексту процесу, який стає активним. Контекст процесу включає вміст усіх регістрів загального призначення процесора і регістр прапорів операцій. Контекст також містить системні регістри і покажчики, які пов'язані з цим процесом, а не з операційною системою, наприклад покажчик на таблицю трансляції адрес процесу. Для зберігання контекстів призупинених процесів використовуються області оперативної пам'яті, які підтримуються покажчиками процесора.

Перемикання контексту виконується за певними командами процесора, наприклад, за командою переходу на нове завдання. Така команда викликає автоматичне завантаження даних зі збереженого контексту в регістри процесора,

4.1.4 Система переривань

Система переривань дозволяє комп'ютеру реагувати на зовнішні події, синхронізувати виконання процесів і роботу пристроїв введення-виведення, швидко переходити з однієї програми на іншу. Механізм переривань потрібний для того, щоб повідомити процесор про виникнення в обчислювальній системі деякої непередбачуваної події.

Прикладами таких подій можуть служити завершення операції введення-виведення зовнішнім пристроєм, некоректне завершення арифметичної операції. При виникненні умов переривання його джерело (контролер зовнішнього пристрою, таймер, арифметичний блок процесора і т.п.) виставляє певний електричний сигнал. Цей сигнал перериває виконання процесором послідовності

команд і викликає автоматичний перехід на заздалегідь визначену процедуру, що називається процедурою обробки переривань.

У більшості моделей процесорів відпрацьовуваний апаратурою перехід на процедуру обробки переривань супроводжується заміною слова стану машини (або навіть усього контексту процесу), що дозволяє одночасно з переходом за потрібною адресою виконати перехід в привілейований режим. Після завершення обробки переривання відбувається повернення до виконання перерваного коду.

Переривання грають найважливішу роль в роботі будь-якої операційної системи, будучи її рушійною силою. Дійсно, велика частина дій ОС ініціюється перериваннями різного типу. Навіть системні виклики від додатків виконуються на багатьох апаратних платформах за допомогою спеціальної інструкції переривання, що викликає перехід до виконання відповідних процедур ядра (наприклад, інструкція int в процесорах Intel або SVC в мейнфреймах IBM).

Переривання бувають двох типів: апаратні і програмні.

Апаратні переривання – це спеціальний сигнал (запит переривання, **IRQ**), який передається процесору від апаратного пристрою. До апаратних переривань належать:

– переривання введення-виведення, які приходять від контролера периферійного пристрою (наприклад, такі переривання генерує контролер клавіатури при натисненні на клавішу);
– переривання, пов'язані з апаратними або програмними помилками (такі переривання виникають, наприклад, у разі збою контролера диска, доступу до захищених областей пам'яті, ділення на нуль).

Програмні переривання генерують програми, що виконують спеціальну інструкцію переривання. Така інструкція є в системі команд процесора. Обробка програмних переривань процесором не відрізняється від обробки апаратних переривань.

Якщо переривання сталося, то процесор передає управління спеціальній процедурі – ***обробнику переривань.*** Після виходу з обробника процесор продовжує виконання інструкцій перерваної програми. Відрізняють два типи програмних переривань залежно від того, яка інструкція буде виконана після виходу з обробника: для **відмови** (faults) повторюється інструкція, яка викликала переривання, для **пасток (**traps) – виконується наступна інструкція.

Усі переривання введення-виведення і програмні переривання належать до категорії пасток.

Для реалізації ***привілейованого режиму процесора*** в одному з його регістрів передбачений спеціальний біт (***біт режиму***), який вказує, в якому режимі працює процесор. У разі апаратного або програмного переривання процесор автоматично перемикається в привілейований режим.

Альтернативою перериванням є періодичне опитування стану кожного пристрою з боку процесора. Подібний підхід, що називається послідовним опитуванням (pooling), підвищує накладні витрати і збільшує складність комп'ютерної системи. Переривання позбавляють процесор від необхідності постійно опитувати системні пристрої.

4.1.5 Системний таймер

Системний таймер, що часто реалізовується у вигляді швидкодіючого регістра-лічильника, потрібний операційній системі для витримки інтервалів часу. Для цього в регістр таймера програмно завантажується значення необхідного інтервалу в умовних одиницях, з якого потім автоматично з певною частотою (при кожному імпульсі кварцевого генератора) починає відніматися по одиниці. Частота «тиків» таймера, як правило, тісно пов'язана з частотою тактового генератора процесора.

Не слід плутати таймер ні з тактовим генератором, який виробляє сигнали, синхронізуючі усі операції в комп'ютері, ні з системним годинником, якій працює на батареях і веде незалежний відлік часу. Досягши нульового значення лічильника таймер ініціює переривання, яке обробляється процедурою операційної системи. Переривання від системного таймера використовуються ОС в першу чергу для стеження за тим, як окремі процеси витрачають час процесора. Наприклад, в системі розподілу часу при обробці чергового переривання від таймера планувальник процесів може примусово передати управління іншому процесу, якщо цей процес вичерпав виділений йому квант часу.

4.1.6 Засоби захисту областей пам'яті

Засоби захисту областей пам'яті забезпечують на апаратному рівні перевірку можливості програмного коду здійснювати з даними певної області пам'яті такі операції, як читання, запис або виконання (при передачах управління). Якщо апаратура комп'ютера підтримує механізм трансляції адрес, то засоби захисту областей пам'яті вбудовуються в цей механізм. Функції апаратури з захисту пам'яті полягають в порівнянні рівнів привілеїв поточного коду процесора і сегменту пам'яті, до якого робиться звернення.

4.2 Машинно-залежні компоненти ОС

Одна і та ж операційна система не може без яких-небудь змін встановлюватися на комп'ютерах, що відрізняються типом процесора або/і способом організації усієї апаратури. У модулях ядра ОС не можуть не відбитися такі особливості апаратної платформи, як кількість типів переривань і формат таблиці посилань на процедури обробки переривань, склад регістрів загального призначення і системних регістрів, стан яких треба зберігати в контексті процесу, особливості підключення зовнішніх пристроїв і багато інших.

Проте досвід розробки операційних систем показує: ядро можна спроектувати таким чином, що тільки частина модулів будуть машинно-залежними, а інші не залежатимуть від особливостей апаратної платформи. У добре структурованому ядрі машинно-залежні модулі локалізовані і утворюють програмний шар, що примикає до шару апаратури. Така локалізація машинно-залежних модулів істотно спрощує перенесення операційної системи на іншу апаратну платформу.

Об'єм машинно-залежних компонентів ОС залежить від того, наскільки великі відмінності в апаратних платформах, для яких розробляється ОС. Наприклад, ОС, побудована на 32-бітових адресах, для перенесення на машину з 16-бітовими адресами має бути практично переписана наново. Проте одна з найочевидніших відмінностей – розбіжність системи команд процесорів – долається досить просто. Операційна система програмується мовою високого рівня, а потім відповідним компілятором виробляється код для конкретного типу процесора.

Проте в багатьох випадках відмінності в організації апаратури комп'ютера лежать набагато глибше і здолати їх таким чином не вдається. Наприклад, однопроцесорний і двопроцесорний комп'ютери вимагають застосування в ОС абсолютно різних алгоритмів розподілу процесорного часу. Аналогічна відсутність апаратної підтримки віртуальної пам'яті призводить до принципової відмінності в реалізації підсистеми управління пам'яттю. У таких випадках не обійтися без внесення в код операційної системи специфіки апаратної платформи, для якої ця ОС належить.

Для зменшення кількості машинно-залежних модулів виробники операційних систем обмежують універсальність машинно-незалежних модулів. Це означає, що їх незалежність носить умовний характер і поширюється тільки на декілька типів процесорів і створених на основі цих процесорів апаратних платформ. По цьому шляху пішли, наприклад, розробники ОС Windows NT, обмеживши кількість типів процесорів для своєї системи чотирма і поставляючи різні варіанти кодів ядра для однопроцесорних і багатопроцесорних комп'ютерів.

Особливе місце серед модулів ядра займають низькорівневі драйвери зовнішніх пристроїв. З одного боку ці драйвери, як і високорівневі драйвери, входять до складу менеджера введення-виведення, тобто належать шару ядра, що займає досить високе місце в ієрархії шарів. З іншого боку, низькорівневі драйвери відбивають усі особливості керованих зовнішніх пристроїв, тому їх можна віднести і до шару машинно-залежних модулів. Така двоїстість низькорівневих драйверів ще раз підтверджує схемну моделі ядра із строгою ієрархією шарів.

Для комп'ютерів на основі процесорів Intel x86/Pentium розробка екрануючого машинно-залежного шару ОС дещо спрощується за рахунок вбудованої в постійну пам'ять комп'ютера базової системи введення-виведення – **BIOS**. BIOS містить драйвери для усіх пристроїв, що входять в базову конфігурацію комп'ютера: жорстких і гнучких дисків, клавіатури, дисплея тощо.

Ці драйвери виконують дуже примітивні операції з керованими пристроями, наприклад читання групи секторів даних з певної доріжки диска, але за рахунок цих операцій екрануються відмінності апаратних платформ персональних комп'ютерів і серверів на процесорах Intel різних виробників. Розробники операційної системи можуть користуватися шаром драйверів BIOS як частиною машинно-залежного шару ОС.

В ідеалі шар машинно-залежних компонентів ядра повністю екранує іншу частину ОС від конкретних деталей апаратної платформи (кеші, контролери переривань введення-виведення тощо), принаймні для того набору платформ,

який підтримує ця ОС. У результаті відбувається підміна реальної апаратури деякою уніфікованою віртуальною машиною, однаковою для усіх варіантів апаратної платформи. Усі шари операційної системи, які лежать вище шару машинно-залежних компонентів, можуть бути написані для управління саме цією віртуальною апаратурою. Таким чином, у розробників з'являється можливість створювати один варіант машинно-незалежної частини ОС (включаючи компоненти ядра, утиліти, системні оброблювальні програми) для всього набору підтримуваних платформ.

4.3 Основні елементи комп'ютера

На макрорівні комп'ютер складається з процесора, пам'яті і пристроїв введення-виведення. При цьому кожен компонент представлений одним або декількома модулями. Щоб комп'ютер міг виконувати своє основне призначення, що полягає у виконанні програм, різні компоненти повинні мати можливість взаємодіяти між собою. Можна виділити чотири структурні компоненти комп'ютера:

1. **Процесор**. Здійснює контроль за діями комп'ютера, а також виконує функцію обробки даних.
2. **Основна пам'ять**. Тут зберігаються дані і програми.
3. **Пристрої введення-виведення.** Служать для передачі даних між комп'ютером і зовнішнім оточенням, що складається з різних периферійних пристроїв.
4. **Системна шина**. Певні структури і механізми, що забезпечують взаємодію між процесором, основною пам'яттю і пристроями введення-виведення.

4.4 Процесори

«Мозком« комп'ютера є **центральний процесор** (**CPU** – Central Processing Unit). Звичайний цикл роботи центрального процесора виглядає так: він читає першу команду з пам'яті, декодує її для визначення типу і операндів команди, виконує команду, потім прочитує, декодує і виконує подальші команди. Таким чином здійснюється виконання програм.

4.4.1 Регістри процесора

Для кожного процесора існує свій набір команд, які він в змозі виконати. Оскільки доступ до пам'яті для отримання команд або набору даних займає набагато більше часу, ніж виконання цих команд, усі процесори містять внутрішні регістри для зберігання ключових змінних і тимчасових результатів. Тому набір інструкцій для будь-якого процесора містить команди для завантаження слова з пам'яті в регістр і збереження слова з регістра в пам'ять. Інші команди об'єднують два операнди з регістрів, пам'яті або того і іншого і отримують результат. Регістри процесора виконують дві функції:

1. **Регістри управління і регістри стану**. Використовуються в процесорі для контролю над виконуваними операціями. За їх допомогою привілейовані програми ОС можуть контролювати хід виконання інших програм.
2. **Регістри, доступні користувачеві**. Ці регістри дозволяють програмістові скоротити число звернень до основної пам'яті, оптимізуючи використання регістрів за допомогою мови асемблера або мови високого рівня.

Регістри управління і регістри стану. Однією з функцій процесора є обмін даними з пам'яттю. Для цього використовуються два внутрішні регістри процесора: ***регістр адреси пам'яті*** (memory address register – **MAR)**, куди заноситься адреса елементу пам'яті, в якій робитиметься операція читання-запису, і ***регістр буфера пам'яті*** (memory buffer register **– MBR)**, куди заносяться дані, призначені для запису в пам'ять. Аналогічно номер облаштування введення-виведення задається в ***регістрі адреси введення-виведення*** (I/O address register **– I/O AR**). Регістр ***буфера введення-виведення*** (I/O buffer register **– I/O BR**).

Окрім цих регістрів є ще декілька спеціальних регістрів. Один з них називається ***лічильником команд*** (program counter **– PC**), у ньому міститься адреса наступної команди, що стоїть у черзі на виконання. Після того, як команда вибрана з пам'яті, регістр команд коригується і покажчик переходить до наступної команди. ***Регістр команд*** (instruction register – **IR**) містить останню вибрану в пам'яті команду.

Наступний регістр (чи набір регістрів) називається **PSW** (Processor Status Word – ***слово стану процесора***). Цей регістр містить коди умов і біти коду станів, які задаються командами порівняння, пріоритетом процесора, режимом (режим користувача або режим ядра), та іншу службову інформацію.

Коди умов (відомі також як ***прапори***) – це послідовність бітів, що встановлюються або скидаються процесором залежно від результату виконання операцій. Наприклад, у результаті виконання арифметичної дії може вийти від'ємне число, нуль, або переповнювання. У результаті арифметичних операцій встановлюються також відповідні коди умов. Потім вони можуть бути перевірені умовною операцією галуження. Призначені для користувача програми за допомогою спеціальних команд можуть читати увесь регістр PSW цілком, але писати можуть тільки в деякі з його полів. Регістр PSW відіграє важливу роль в системних викликах і операціях уведення-виведення.

Регістри, доступні користувачеві. До цих регістрів користувач може звертатися за допомогою команд машинної мови. Серед доступних регістрів є регістри даних, адресні регістри і регістри коду умови.

Регістри даних можна застосовувати в різних цілях. Проте при цьому накладаються певні обмеження. Наприклад, деякі регістри призначені для операцій над числами з плаваючою точкою, тоді як інші – для зберігання цілих чисел.

Адресні регістри призначені для занесення адрес команд і даних в основній пам'яті. У цих регістрах може бути записана тільки частина адреси, що використовується при обчисленні повної адреси.

Ще один регістр процесора називається ***покажчиком стека*** (**SP**, stack pointer). Він містить адресу вершини стека в пам'яті. Стек містить по одному ***фрейму*** (області даних) для кожної процедури, яка вже почала виконуватися, але ще не закінчилася. У стековому фреймі процедури зберігаються її вхідні параметри, а також локальні і тимчасові змінні, що не зберігаються в регістрах. У деяких машинах виклик процедури або підпрограми призводить до автоматичного збереження вмісту усіх доступних користувачеві регістрів, щоб після повернення їх можна було відновити.

Регістри процесора Pentium. Оскільки в основному в ПК використовуються процесори типу Pentium, то розглянемо їх регістри детальніше. У процесорах Pentium ці регістри діляться на декілька груп:

- регістри загального призначення;
- регістри сегментів;
- покажчик інструкцій;
- регістр прапорів;
- управляючі регістри;
- регістри системних адрес;
- регістри відладки і тестування;
- регістри математичного співпроцесора, що виконує операції з плаваючою точкою.

У процесорі Pentium є вісім 32-розрядних регістрів загального призначення. Чотири з них, які можна умовно назвати A, B, C і D, використовуються для тимчасового зберігання операндів арифметичних, логічних і інших команд. Програміст може звертатися до цих регістрів як до єдиного цілого, використовуючи позначення EAX, EBX, ECX, EDX, а також до деяких їх частин, як це показано на рис. 4.1. Тут позначення AL(L – Low) стосується першого, наймолодшого байта регістра EAX, AH (H – High) – до наступного за старшинством байта, а AX означає обидва молодші байти регістра. Приставка E в позначенні цих регістрів утворена від слова ***extended (розширений***), що вказує на те, що в колишніх моделях процесорів Intel ці регістри були 16-розрядними, а потім їх розрядність була збільшена до 32 біт.

Інші чотири регістри загального призначення – ESI, EDI, EBP і ESP – призначені для задання зміщення адреси відносно початку деякого сегменту даних. Ці регістри використовуються спільно з регістрами сегментів в системі адресації процесора Pentium для задання віртуальної адреси, яка потім за допомогою таблиць сторінок відображається на фізичну адресу.

Регістри сегментів CS, SS, DS, ES, FS і GS в захищеному режимі посилаються на дескриптори сегментів пам'яті – описувачі, в яких містяться такі параметри сегментів, як базова адреса, розмір сегменту, атрибути захисту і деякі інші. Регістри сегментів зберігають 16-розрядне число, що називається селектором, в якому 12 старших розрядів є індексом в таблиці дескрипторів сегментів, 1 розряд вказує, в якій з двох таблиць, GDT або LDT, знаходиться дескриптор, а три розряди поля RPL зберігають значення рівня привілеїв запиту до цього сегменту.

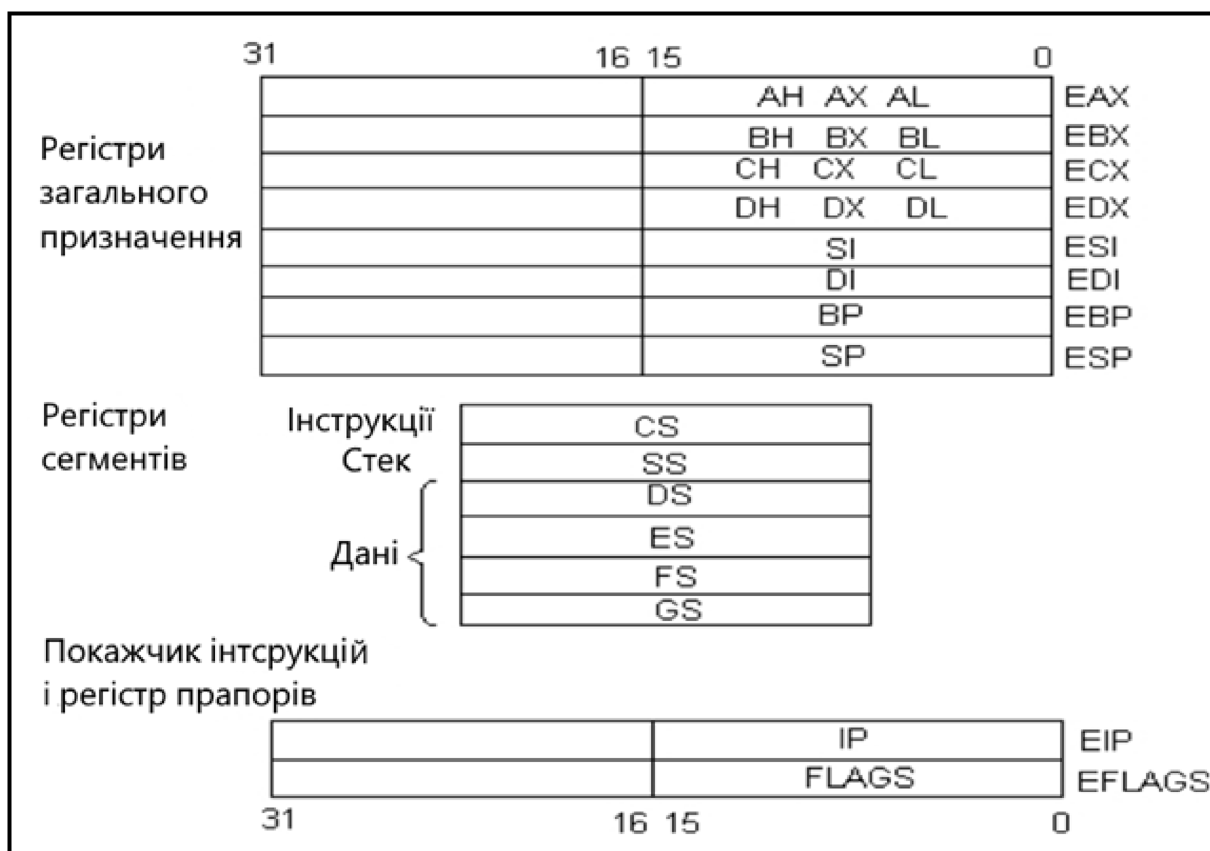


Рисунок 4.1 – Основні регістри процесора Pentium

Регістр CS (Code Segment) призначений для зберігання індексу дескриптора кодового сегменту, регістр SS (Stack Segment) – дескриптора сегменту стека, а інші регістри використовуються для вказівки на дескриптори сегментів даних. Усі регістри сегментів, окрім CS, програмно доступні, тобто в них можна завантажити нове значення селектора відповідною командою (наприклад, LDS). Значення регістра CS змінюється при виконанні команд міжсегментних викликів CALL і переходів JMP, а також при перемиканні задач.

У цьому розділі термін «задача» часто вживатиметься замість рівнозначного (і поширенішого) терміну «процес» у зв'язку з тим, що саме цей термін вибрали свого часу розробники процесорів Intel x86, і він фігурує в назвах регістрів і структур даних.

Покажчик інструкцій EIP містить зміщення адреси поточної інструкції, яке використовується спільно з регістром CS для отримання відповідної віртуальної адреси.

Регістр прапорів EFLAGS містить ознаки, що характеризують результат виконання операції, наприклад, прапор знаку, прапор нуля, прапор переповнювання, прапор паритету, прапор перенесення і деякі інші. Крім того, тут зберігаються деякі ознаки, що встановлюються і аналізуютсься механізмом переривань, зокрема прапор дозволу апаратних переривань IF.

У процесорі Pentium є п'ять керуючих регістрів – CRO, CR1, CR2, CR3 і CR4, які зберігають ознаки і дані, що характеризують загальні стани процесора (рис. 4.2).

Рисунок 4.2 – Керуючі і системні регістри процесора Pentium

Регістр CR0 містить усі основні ознаки, що істотно впливають на роботу процесора, такі як реальний/захищений режим роботи, включення/виключення сторінкового механізму системи віртуальної пам'яті, а також ознаки, що впливають на роботу кеша і виконання команд з плаваючою точкою. Молодші два байти регістра CRO мають назву Mashine State Word, MSW – «слово стану машини». Ця назва використовувалася в процесорі 80286 для позначення куруючого регістра, який мав аналогічне призначення.

Регістр CR1 нині не використовується (зарезервований).

Регістри CR2 і CR3 призначені для підтримки роботи системи віртуальної пам'яті. Регістр CR2 містить лінійну віртуальну адресу, яка викликала так звану сторінкову відмову (відсутність сторінки в оперативній пам'яті або відмова із-за порушення прав доступу). Регістр CR3 містить фізичну адресу таблиці розділів, яка використовується сторінковим механізмом процесора.

У регістрі CR4 зберігаються ознаки, що дозволяють роботу архітектурних розширень, наприклад, можливості використання сторінок розміром 4 Мб тощо.

Регістри системних адрес містять адреси важливих системних таблиць і структур, використовуваних при управлінні процесами і пам'яттю. Регістр GDTR (Global Descriptor Table Register) містить фізичну 32-розрядну адресу глобальної таблиці дескрипторів GDT сегментів пам'яті, що утворюють загальну частину віртуального адресного простору усіх процесів.

Регістр IDTR (Interrupt Descriptor Table Register) зберігає фізичну 32-розрядну адресу таблиці дескрипторів переривань IDT, використовувану для виклику процедур обробки переривань у захищеному режимі роботи процесора. Окрім цих адрес у регістрах GDTR і IDTR зберігаються 16-бітові ліміти, що задають обмеження на розмір відповідних таблиць.

Два 16-бітові регістри зберігають не фізичні адреси системних структур, а значення індексів дескрипторів цих структур в таблиці GDT, що дозволяє побічно отримати відповідні фізичні адреси. Регістр TR (Task Register) містить індекс дескриптора сегменту стану задачі TSS. Регістр LDTR (Local Descriptor Table Register) містить індекс дескриптора сегменту локальної таблиці дескрипторів LDT сегментів пам'яті, що утворюють індивідуальну частину віртуального адресного простору процесу. Регістри відладки зберігають значення точок останову, а регістри тестування дозволяють перевірити коректність роботи внутрішніх блоків процесора.

4.4.2 Виконання команд

З метою покращення характеристик процесорів їх розробники давно відмовилися від простої моделі, в якій за один такт може бути зчитана, декодована і виконана тільки одна команда. Багато сучасних процесорів мають можливості виконання декількох команд одночасно. Наприклад, у процесора можуть бути роздільні модулі, що займаються вибіркою, декодуванням і виконанням команд. Під час виконання команди з номером *n* він може декодувати команду з номером *n+1* і прочитувати команду з номером *n+2*. Подібна організація процесу називається ***конвеєром*** (рис. 4.3, *а*) [9].

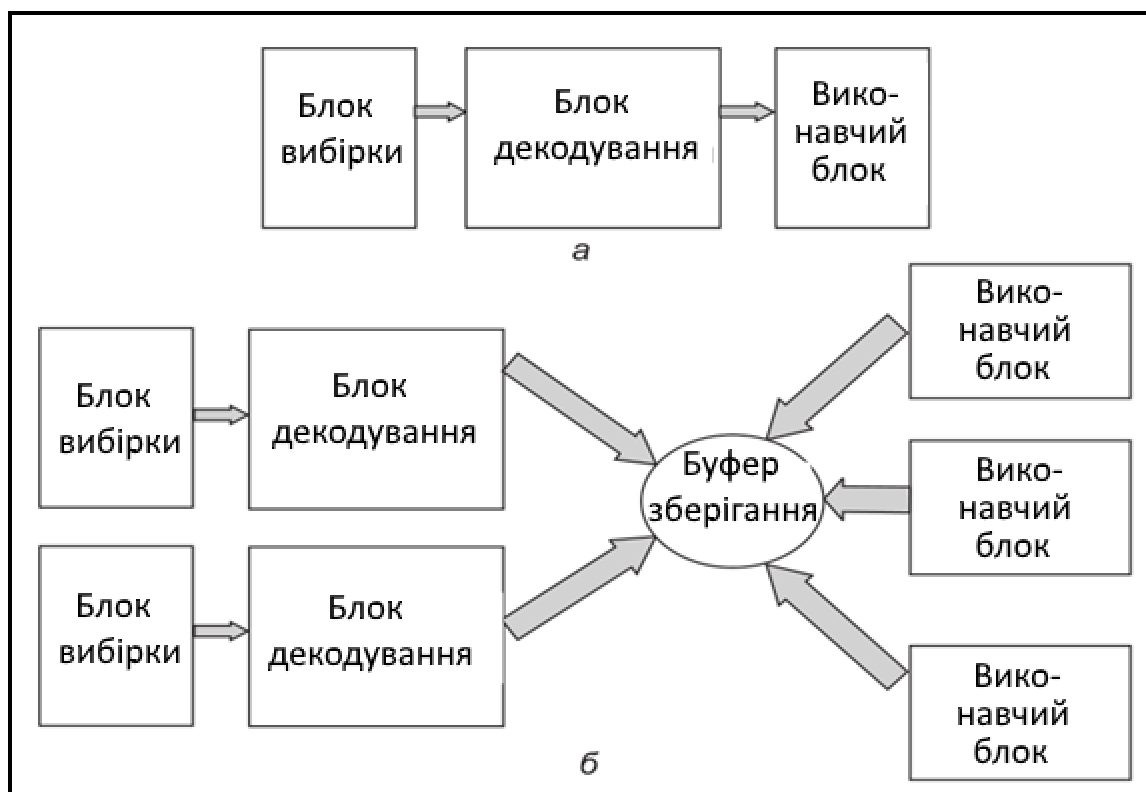


Рисунок 4.3 – Процесор: *а* – з конвеєром; *б* – суперскалярний

Часто зустрічаються і довші конвеєри. У більшості конвеєрних конструкцій зчитана команда має бути виконана, навіть якщо в попередній команді був прийнятий умовний перехід. Ця властивість конвеєрів є проблемою для розробників компіляторів і операційних систем.

Передовішим у порівнянні з конвеєрною конструкцією є ***скалярний процесор*** (см. рис. 4.3, *б*). У цій структурі є присутньою певна кількість виконнавчих вузлів: один для цілочисельних арифметичних операцій, другий – для операцій з плаваючою точкою і ще один – для логічних операцій. За один такт прочитується дві або більше команди, які декодуються і скидаються в буфер зберігання, де вони чекають своєї черги на виконання. Коли виконуючий пристрій звільняється, він «заглядає» в буфер зберігання, і якщо там є команда, яку воно може обробити, то забирає її і виконує. У результаті команди часто виконується не в порядку їх слідування. Проте при цьому підході дуже неприємні ускладнення торкнулися і ОС.

Якщо зупинитися на найскладнішому елементі комп'ютера, що визначає технічний рівень усього виробу, – центральному процесорі – то можна помітити, що прогрес тут йде двома паралельними шляхами: розвиток елементної бази і вдосконалення архітектури**.**

Розвиток елементної бази. Технологія виготовлення мікропроцесорів розвивається в напрямі подальшої мініатюризації електронних схем і, як наслідок, підвищення ступеня їх інтеграції. Зменшення розмірів дозволяє «упакувати» на одному чипі більше число елементів і ускладнити мікросхему. Ще в 1968 році, на зорі мікроелектроніки, один із засновників фірми Intel **Гордон Мур** сформулював емпіричний «закон Мура», за яким число елементів на одному кристалі повинне подвоюватися кожні півтора роки. Дивно, але факт – пройшло більше 48 років, невпізнанно змінилися технології, проте закон продовжує діяти і зараз.

У повній відповідності із законом Мура, сучасні мікропроцесори є неймовірно складними пристроями. Наприклад, кристал P5 фірми Intel, випущений у 1993 році, що отримав торгову марку «Pentium», містить близько 3 млн транзисторів, P6 – «Pentium Pro» (1996 р.) – 7,5 млн, а процесор P7 (під час розробки він називався «Merced», а в продажу пішов під ім'ям «Itanium»), випуск якого почався в 2000 році, мав близько 20 млн транзисторів.

Зменшення розмірів деталей і довжин провідників, що з'єднують їх, дозволяє удосконалити ще одну характеристику мікропроцесора, що пропорційно впливає на його продуктивність – тактову частоту. Якщо у чіпа i4004 вона дорівнювала 108 кГц, то сучасні схеми допускають збільшення тактової частоти до 2000 – 3000 Мгц.

Удосконалення архітектури. На жаль, нескінченно зменшувати розміри елементарних схем перемикачів неможливо, оскільки вони обмежені знизу розмірами кристалічних решіток. Так само не можна безмежно підвищувати тактову частоту, оскільки швидкість поширення електричного струму кінцева.

Мабуть, найближчим часом елементні можливості мікросхем підійдуть до теоретичної межі, подальше підвищення продуктивності комп'ютерів досягатиметься тільки за рахунок удосконалення архітектури, яке розвивається в чотирьох основних напрямах.

1. Збільшення розрядності. Тенденція до підвищення розрядності виразно простежується в історії мікропроцесорів. Сучасні кристали в основному 32-розрядні, проте просунуті мікросхеми, наприклад, PowerPC, а також

перспективні масові моделі, наприклад, «Merced» – 64-розрядні. Мабуть, у майбутньому можна очікувати і появи 128-бітових чіпів.

2. Рух у напрямку RISC. Абревіатура RISC розшифровується як Reduced Instruction Set Computer – комп'ютер із скороченим набором команд.

Для того щоб зрозуміти сенс цього явища, треба повернутися до ранньої історії ЕОМ. В ті часи алгоритмічні мови і компілятори ще не були відомі, і усе програмування велося вручну, в командах процесора. Тому розробники комп'ютерів намагалися зробити систему команд зручною для ручного програмування, наповнивши її складними і місткими командами.

Наприклад, однією машинною командою можна було обчислити функцію log або sin, або перетворити число в іншу систему числення. Репертуар машинних команд виходив досить складним. Наприклад, в IBM-360 були реалізовані 144 команди центрального процесора. Така організація системи команд дістала назву **CISC** – Complex Instruction Set Computing, тобто обчислення зі складним набором команд.

Основоположником CISC-архітектуру можна вважати компанію IBM з її базовою архітектурою /360, ядро якої використовується з 1964 року і дійшло до наших днів. Лідером в розробці мікропроцесорів з повним набором команд CISC вважається компанія Intel зі своєю серією x86 і Pentium (виключаючи сучасні Intel Pentium 4, Pentium D, Core, AMD Athlon, Phenom, які є гібридними), а також процесори Motorola MC680x0. Ця архітектура є практичним стандартом для ринку мікрокомп'ютерів.

Для CISC-процесорів характерно:

- порівняно невелике число регістрів загального призначення;
- велика кількість машинних команд, деякі з яких навантажені семантично аналогічно операторам високорівневих мов програмування і виконуються за багато тактів;
- велика кількість методів адресації;
- велика кількість форматів команд різної розрядності;
- переважання двоадресного формату команд;
- наявність команд обробки типу регістр-пам'ять.

Стандартний набір команд чіпа i8086 і усіх подальших поколінь процесорів Intel містить близько ста інструкцій найрізноманітнішого призначення і формату. Оскільки формат команди змінний, то вона може бути коректно вибрана з пам'яті тільки після розшифровки коду операції. У результаті кожна інструкція вимагає для свого виконання декілька тактів процесора. Програма, що реалізовує деякий алгоритм, може бути відносно короткою, проте час виконання цієї програми в комп'ютері виявляється значним.

Процесори з **RISC-**архітектурою працюють по-іншому. У них набір команд сильно обмежений, усі інструкції максимально спрощені, вони мають однаковий формат і, в ідеалі, можуть виконуватися за один машинний такт. Програма, що виконує той же алгоритм примітивними командами, виходить довшою, проте за рахунок високої швидкодії процесора спостерігається значний виграш в продуктивності.

Зрозуміло, що програмувати вручну для такої машини було б незручно, проте цього ніхто і не робить, оскільки техніка компіляції досягла великих висот. Швидкодіючі оптимізуючі компілятори дозволяють створити такий код, який використовує усі особливості набору команд і дозволяє добитися найвищої обчислювальної потужності.

Зачатки цієї RISC-архітектури йдуть своїми коренями до комп'ютерів CDC6600, розробники яких (Торнтон, Крей та ін.) усвідомили важливість спрощення набору команд для побудови швидких обчислювальних машин. Цю традицію спрощення архітектури С. Крей з успіхом застосував при створенні широко відомої серії суперкомп'ютерів компанії Cray Research. Проте остаточне поняття RISC-процесорів в сучасному його розумінні сформувалося на початку 1980-х років на базі трьох дослідницьких проектів університету Берклі, Стенфордському і Каліфорнійському університетах США. Процесори 801 компанії IBM, процесори RISC університету Берклі і процесори MIPS Стенфордського університету виконували невеликий (50-100) набір команд, тоді як звичайні CISC-процесори виконували 100-200 команд.

Прибічники RISC-архітектури на ділі довели силу своїх аргументів – найпродуктивніші сервери і робочі станції сьогодні використовують RISC-процесори. Проте і прихильники CISC-технології не здаються, на їх боці велетенський об'єм накопиченого програмного забезпечення в кодах ix86. В останніх моделях мікропроцесорів Intel спеціально для мультимедійних додатків введені ще складніші «векторні» команди додаткового набору MMX (MultiMedia eXtention – мультимедійне розширення), що виконують в наддовгих (128 розрядів) регістрах паралельно декілька операцій складання або множення.

Характерні особливості RISC-процесорів.

1. Фіксована довжина машинних інструкцій (наприклад, 32 біти) і простий формат команди.
2. Спеціалізовані команди для операцій з пам'яттю – читання або запису. Операції виду «прочитати-змінити-записати» відсутні. Будь-які операції «змінити» виконуються тільки над вмістом регістрів.
3. Велика кількість регістрів загального призначення (32 і більше в порівнянні з 8-16 регістрами в CISC архітектурі), що дозволяє великому об'єму даних зберігатися в регістрах на процесорному кристалі більший час і спрощує роботу компілятора з розподілу регістрів під змінні.
4. Відсутність підтримки операцій виду «змінити» над укороченими типами даних – байт, 16-бітове слово. Так, наприклад, система команд DEC Alpha містила тільки операції над 64-бітовими словами, і вимагала розробки і подальшого виклику процедур для виконання операцій над байтами, 16- і 32-бітовими словами.
5. Відсутність мікропрограм усередині самого процесора. Те, що в CISC процесорі виконується мікропрограмами, в RISC процесорі виконується як звичайний (хоча і поміщений в спеціальне сховище) машинний код, що не відрізняється принципово від коду ядра ОС і додатків.

Для того щоб об'єднати переваги обох підходів, в останніх розробках компанії Intel (маються на увазі Pentium і Pentium Pro), а також її послідовників-

конкурентів (AMD R5, Cyrix M1, NexGen Nx586 та ін.) використовуються ідеї, реалізовані в RISC-мікропроцесорах, на ходу перетворюючи CISC команди в набір RISC команд і виконуючи їх на своєму RISC ядрі (***гібридна архітектура***). На зовнішньому рівні мікропроцесор виконує стандартний CISC-набір команд, а на внутрішньому – деякий спрощений RISC. Вбудований мікропрограмний емулятор перетворює кожну зовнішню команду в ланцюжок внутрішніх, і виконує її з усією можливою продуктивністю RISC-обчислювача.

3. Ускладнення архітектури процесора. Ще один резерв підвищення продуктивності криється в розпаралелюванні обчислень усередині одного кристала, при цьому розробники мікросхем намагаються реалізувати в конструкції принципи, типічні для організації промислового виробництва.

Як відомо, кожна машинна операція складається з декількох фаз: вибірка команди, розшифровка її, читання операндів, безпосереднє виконання операції, запис результату. У старих моделях процесора ці фази виконувалися для кожної операції строго послідовно подібно до того, як в кустарних майстернях йшло колись складання автомобілів – спочатку збирали одну машину, потім другу, при цьому частина робітників постійно простоювала.

Сучасний мікропроцесор влаштований значно складніше. Він схожий на підприємство, в якому величезна кількість робітників збирає на конвеєрі потік автомобілів. Конвеєрний процесор поєднує за часом виконання декількох команд: для однієї відбувається читання операції, для другої – декодування і вибірка регістрів, для третьої – виконання команди обчислювальним блоком тощо. В результаті при тій же тактовій частоті істотно підвищується загальна продуктивність. Більше того, в найдосконаліших конструкціях в чіп мікропроцесора вбудовується декілька самостійних (до 6-8) обчислювальних блоків з фіксованою і плаваючою арифметикою, надшвидка внутрішня пам'ять (кеш) і управляючий пристрій. Неминуча плата за таку організацію – значне підвищення складності і вартості схеми. Проте прогрес мікроелектроніки дозволив реалізувати таку архітектуру в усіх сучасних моделях мікропроцесорів.

4. Багатопроцесорні конфігурації. Коли можливості одного кристала вичерпані, продуктивність комп'ютера в цілому може бути збільшена за рахунок багатопроцесорної організації. Аналіз реальних додатків показує, що довгі ланцюжки машинних команд, які повинні виконуватися строго послідовно, зустрічаються відносно нечасто, в основному в наукових розрахунках.

У принципі число процесорів в комп'ютері нічим не обмежене. Відомі конструкції з сотнями і навіть тисячами процесорів. Проте сумарна продуктивність багатопроцесорної системи росте далеко не лінійно з числом процесорів, оскільки в кожній програмі є деяка межа розпаралелювання, до того ж у багатопроцесорних системах різко зростають накладні витрати на диспетчеризацію обчислювального процесу.

Практика показала, що на стандартних комерційних задачах продуктивність системи росте як sqrt(n), тобто чотирипроцесорна конфігурація всього в два рази продуктивніше однопроцесорної. Проте на спеціальних задачах, що допускають багатократне розпаралелювання, багатопроцесорні комп'ютери можуть показувати рекорди продуктивності.

4.5 Кешування даних

4.5.1 Ієрархія запам'ятовуючих пристроїв

Пам'ять обчислювальної машини є ієрархією запам'ятовуючих пристроїв (внутрішні регістри процесора, різні типи надоперативної і оперативної пам'яті, диски), які відрізняються середнім часом доступу і вартістю зберігання даних з розрахунку на один біт (рис 4.4) [10].

Фундаментом цих пристроїв служить зовнішня пам'ять, що, як правило, представляється жорстким диском, який частенько ще називають вінчестер. Вона має великий об'єм (сотні і тисячі гігабайт), але швидкість доступу до даних є невисокою. Час доступу до диска вимірюється мілісекундами.

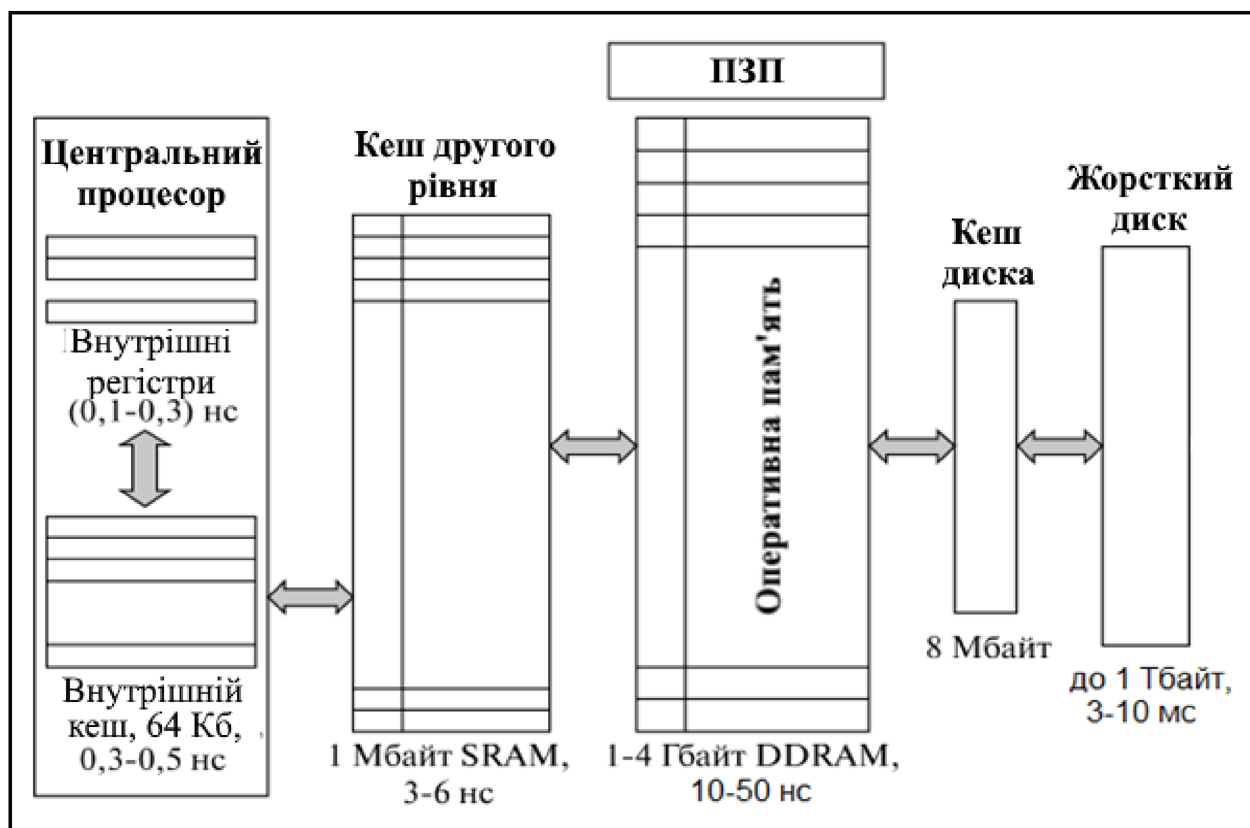


Рисунок 4.4 – Ієрархія пам′яті

Принципи сучасної технології виготовлення жорсткого диска були розроблені в 1973 році американською фірмою IBM. Новий пристрій, який міг зберігати до 16 кілобайт інформації, мав 30 циліндрів (доріжок) для запису, кожен з яких був розбитий на 30 секторів. Тому пристрій одержав назву 30/30. Відомі рушниці «вінчестер» мають калібр 30/30, тому, за однією з версій, жорсткі диски теж стали називатися «військовим» словом – «вінчестерами». За іншою, ймовірнішою версією (Велика енциклопедія Кирила і Мефодія, 2005), він одержав таку назву за містом Вінчестер (Winchester, Англія), де проходили первинні розробки жорсткого диска.

На наступному рівні розташовується більше швидкодіюча (час доступу дорівнює приблизно 10-20 наносекундам) і менш об'ємна (від десятків Мегабайт

до декількох Гігабайт) оперативна пам'ять, що реалізовується на відносно повільній динамічній пам'яті DRAM. Її часто називають **ОЗП** (Оперативний Запам'ятовуючий Пристрій, в англомовній літературі **RAM** – Random Access).

Для зберігання даних, до яких необхідно забезпечити швидкий доступ, використовуються компактні швидкодіючі пристрої на основі статичної пам'яті SRAM, об'єм яких складає від декількох десятків до декількох сотень кілобайт, а час доступу до даних не перевищує 6 нс.

І нарешті, верхівку в цій ієрархії складають внутрішній кеш і внутрішні регістри процесора, які також можуть бути використані для проміжного зберігання даних. Загальний об'єм регістрів складає декілька десятків байт, а час доступу визначається швидкодією процесора і рівний нині приблизно 0,1-0,3 нс.

Усі названі характеристики запам'ятовуючих пристроїв швидко змінюються в міру вдосконалення обчислювальної апаратури. У даному випадку важливі не абсолютні значення часу доступу або об'єму пам'яті, а їх співвідношення для різних типів запам'ятовуючих пристроїв.

Таким чином, можна констатувати закономірність – чим більше об'єм запам'ятовуючого пристрою, тим менш швидкодіючим він є. Більше того, вартість зберігання даних з розрахунку на один біт також збільшується із зростанням швидкодії пристроїв. Проте користувачеві хотілося б мати і недорогу і швидку пам'ять. Кеш-пам'ять представляє деяке компромісне рішення цієї проблеми.

4.5.2 Кеш-пам'ять

Кожен з нас використовує кешування у своєму повсякденному житті. У загальних рисах, кеш – це легкодоступне місце для зберігання необхідних речей. Ми зберігаємо олівці, ручки і скріпки для паперів в ящиках нашого письмового столу, отже ми легко можемо їх узяти, коли вони нам знадобляться.

Кеш-пам'ять – це спосіб організації спільного функціонування двох типів запам'ятовуючих пристроїв, які відрізняються часом доступу і вартістю зберігання даних. Цей спосіб дозволяє зменшити середній час доступу до даних за рахунок динамічного копіювання в «швидкий» запам'ятовуючий пристрій (ЗП) інформації з «повільного» ЗП, яка найчастіше використовується.

Кеш-пам'яттю, або кешем, також часто називають не лише спосіб організації роботи двох типів запам'ятовуючих пристроїв, але і один з пристроїв – «швидкий» ЗП. Він коштує дорожче і, як правило, має порівняно невеликий об'єм.

Кешування – це універсальний метод, придатний для прискорення доступу до оперативної пам'яті, до диска і до інших видів запам'ятовуючих пристроїв. Якщо кешування застосовується для зменшення середнього часу доступу до оперативної пам'яті, то в якості кеша використовують швидкодіючу статичну пам'ять. Якщо кешування використовується системою введення-виведення для прискорення доступу до даних, що зберігаються на диску, то в цьому випадку роль кеш-пам'яті виконують буфери в оперативній пам'яті, в яких осідають найактивніше використовувані дані.

4.5.3 Принцип дії кеш-пам'яті

Розглянемо окремий випадок використання кеш-пам'яті для зменшення середнього часу доступу до даних, що зберігаються в оперативній пам'яті. Для цього між процесором і оперативною пам'яттю поміщається швидкий ЗП, такий, що називається просто кеш-пам'яттю (рис. 4.5).

Вміст кеш-пам'яті є сукупністю записів про всі завантажені в неї елементи даних. Кожен запис про елемент даних включає:

- значення елементу даних;
- адреса, яку цей елемент даних має в оперативній пам'яті;
- управляючу інформацію: ознака модифікації і ознака звернення до даних за деякий останній період часу.

Структура кеш-пам'яті

Адреса даних в основній пам'яті	Дані	Керуюча інформація

Рисунок 4.5 – Схема функціонування кеш-пам'яті

У системах, оснащених кеш-пам'яттю, кожен запит до оперативної пам'яті виконується відповідно до такого алгоритму.

1. Проглядається вміст кеш-пам'яті з метою визначення, чи не знаходяться потрібні дані в кеш-пам'яті. Кеш-пам'ять не адресується, тому пошук потрібних даних здійснюється по вмісту – значенню поля «адреса в оперативній пам'яті», взятому із запиту.
2. Якщо дані виявляються в кеш-пам'яті, тобто сталося ***кеш-попадання*** (cache-hit), то вони прочитуються з неї, і результат передається джерелу запиту (наприклад, процесору).
3. Якщо потрібних даних немає в кеш-пам'яті, тобто стався ***кеш-промах*** (cache-miss), то вони разом зі своєю адресою копіюються з оперативної пам'яті в кеш-пам'ять, і результат виконання запиту передається джерелу запиту. При копіюванні даних може виявитися, що в кеш-пам'яті немає

вільного місця, тоді вибираються дані, до яких в останній період було менше всього звернень (чи за допомогою іншого алгоритму), для витіснення їх з кеш-пам'яті.

4. Якщо дані, що витісняються, були модифіковані за час знаходження в кеш-пам'яті, то вони переписуються в оперативну пам'ять. Якщо ж ці дані не були модифіковані, то їх місце в кеш-пам'яті оголошується вільним.

На практиці в кеш-пам'ять прочитується не один елемент даних, до якого сталося звернення, а цілий блок даних. Це збільшує ймовірність так званого ***«попадання в кеш»***, тобто знаходження потрібних даних в кеш-пам'яті.

Інтуїтивно зрозуміло, що ефективність кешування залежить від ймовірності попадання в кеш. Покажемо це шляхом знаходження залежності середнього часу доступу до основної пам'яті від ймовірності кеш-влучень. Нехай є основний запам'ятовуючий, пристрій з середнім часом доступу до даних tl і кеш-пам'ять, що має час доступу $t2$, очевидно, що $t2 < tl$. Нехай t – середній час доступу до даних в системі з кеш-пам'яттю, а p – ймовірність кеш-влучення. За формулою повної ймовірності маємо [19]:

$$\mathbf{t = t1(1 - p) + t2p}.$$

Середній час доступу до даних в системі з кеш-пам'яттю лінійно залежить від ймовірності попадання в кеш і змінюється від середнього часу доступу в основний запам'ятовуючий пристрій $t1$ при $p=0$ до середнього часу доступу безпосередньо в кеш-пам'ять $t2$ при $p=1$. Звідси видно, що використання кеш-пам'яті має сенс тільки при високій ймовірності кеш-влучення.

Ймовірність виявлення даних в кеші залежить від різних чинників, таких, наприклад, як обсяг кешу, обсяг кешованої пам'яті, алгоритму заміщення даних в кеші, особливості виконуваної програми, час її роботи, рівень мультипрограмування і інших особливостей обчислювального процесу. Проте в більшості реалізацій кеш-пам'яті відсоток кеш-влучень виявляється дуже високим – понад 90%. Таке високе значення ймовірності знаходження даних в кеш-пам'яті пояснюється наявністю в них об'єктивних властивостей: просторової і часової локальності.

Просторова локальність. Якщо сталося звернення за деякою адресою, то з високою ймовірністю найближчим часом станеться звернення до сусідніх адрес.

Часова локальність. Якщо сталося звернення за деякою адресою, то наступне звернення за цією ж адресою з великою ймовірністю станеться найближчим часом.

Саме грунтуючись на властивості часової локальності, дані, тільки що зчитані з основної пам'яті, розміщують в пристрої швидкого доступу, припускаючи, що скоро вони знову знадобляться. Зпочатку роботи системи, коли кеш-пам'ять ще порожня, майже кожен запит до основної пам'яті виконується «за повною програмою»: перегляд кеша, констатація промаху, читання даних з основної пам'яті, передача результату джерелу запиту і копіювання даних в кеш. Потім, у міру заповнення кеша, у повній відповідності з властивістю часової локальності зростає ймовірність звернення до даних, які вже були використані

на попередньому етапі роботи системи, тобто до даних, які містяться в кеші і можуть бути зчитані значно швидше, ніж з основної пам'яті.

Властивість просторової локальності також використовується для збільшення ймовірності кеш-попадання. Як правило, в кеш-пам'ять прочитується не один інформаційний елемент, до якого сталося звернення, а цілий блок даних, розташованих в основній пам'яті в безпосередній близькості з цим елементом. Оскільки при виконанні програми дуже висока ймовірність, що команди вибираються з пам'яті послідовно одна за одною з сусідніх комірок, то має сенс завантажувати в кеш-пам'ять цілий фрагмент програми. Аналогічно якщо програма веде обробку деякого масиву даних, то її роботу можна прискорити, завантаживши в кеш частину або навіть увесь масив даних.

4.5.4 Проблема узгодження даних

У процесі роботи вміст кеш-пам'яті постійно оновлюється, тобто час від часу дані з неї повинні витіснятися. Витіснення означає або просте оголошення вільної відповідної області кеш-пам'яті (скидання біта дійсності), якщо дані, що витісняються, за час знаходження в кеші не були змінені, або копіювання даних з кешу в основну пам'ять, якщо вони були модифіковані.

Алгоритм заміни даних в кеш-пам'яті істотно впливає на її ефективність. В ідеалі такий алгоритм повинен, по-перше, бути максимально швидким, щоб не уповільнювати роботу кеш-пам'яті, а, по-друге, забезпечувати максимально можливу ймовірність кеш-влучень.

Наявність в комп'ютері двох копій даних – в основній пам'яті і в кеші – породжує проблему узгодження даних. Якщо відбувається запис в основну пам'ять за деякою адресою, а вміст цієї пам'яті знаходиться в кеші, то в результаті відповідний запис в кеші стає недостовірним. Розглянемо два підходи до вирішення цієї проблеми:

1. ***Наскрізний запис*** (write through). При кожному запиті до основної пам'яті, у тому числі і при запису, проглядається кеш. Якщо дані за запрошеною адресою відсутні, то запис виконується тільки в основну пам'ять. Якщо ж дані, до яких виконується звернення, знаходяться в кеші, то запис виконується одночасно в кеш і основну пам'ять.
2. ***Зворотний запис*** (write back). При виникненні запиту до пам'яті виконується перегляд кеша, і якщо запрошених даних там немає, то запис виконується тільки в основну пам'ять. В іншому ж випадку запис проводиться тільки в кеш-пам'ять. При цьому в описувачі даних робиться спеціальна відмітка (ознака модифікації), яка вказує на те, що при витісненні цих даних з кеша необхідно переписати їх в основну пам'ять, щоб актуалізувати застарілий вміст основної пам'яті.

У деяких алгоритмах заміщення передбачається першочергове вивантаження модифікованих даних. Модифіковані дані можуть вивантажуватися не лише при звільненні місця в кеш-пам'яті для нових даних, але і в «фоновому режимі», коли система не дуже завантажена.

4.5.5 Способи відображення основної пам'яті на кеш

Алгоритм пошуку і алгоритм заміщення даних в кеші безпосередньо залежать від того, яким чином основна пам'ять відображається на кеш-пам'ять. Принцип прозорості вимагає, щоб правило відображення основної пам'яті на кеш-пам'ять не залежало від роботи програм і користувачів. При кешуванні даних широко використовуються дві основні схеми відображення: випадкове відображення і детерміноване відображення.

При ***випадковому відображенні*** елемент оперативної пам'яті в загальному випадку може бути розміщений в довільному місці кеш-пам'яті. Для того щоб надалі можна було знайти потрібні дані в кеші, вони поміщаються туди разом зі своєю адресою, тобто з тією адресою, яку дані мають в оперативній пам'яті. При кожному запиті до оперативної пам'яті виконується пошук в кеші, причому критерієм пошуку виступає адреса оперативної пам'яті із запиту. Очевидна схема простого перебору для пошуку потрібних даних у разі кеша виявляється непридатною із-за неприпустимо великих часових витрат. Для кешів з випадковим відображенням використовується так званий ***асоціативний пошук,*** при якому порівняння виконується не послідовно з кожним записом кеша, а паралельно з усіма його записами (рис. 4.6).

Рисунок 4.6 – Асоціативний пошук в кеші з випадковим відображенням

Ознака, за якою виконується порівняння, називається ***тегом*** (tag). В даному випадку тегом є адреса даних в оперативній пам'яті. Електронна реалізація такої схеми призводить до здорожчання пам'яті, причому вартість істотно зростає зі збільшенням об'єму пристрою кеша. Тому асоціативна кеш-пам'ять використовується в тих випадках, коли для забезпечення високого відсотка попадання достатньо невеликого об'єму пам'яті.

У кешах, побудованих на основі випадкового відображення, витіснення старих даних відбувається тільки в тому випадку, коли вся кеш-пам'ять заповнена і немає вільного місця. Вибір даних на вивантаження здійснюється серед усіх записів кеша. Цей вибір грунтується на тих же прийомах, що і в алгоритмах заміщення сторінок, наприклад вивантаження даних, до яких найдовше не було звернень, або даних, до яких було менше всього звернень.

Детермінований спосіб відображення припускає, що будь-який елемент основної пам'яті завжди відображається в одне і те ж місце кеш-пам'яті. У цьому випадку кеш-пам'ять розділена на рядки, кожний з яких призначений для зберігання одного запису про один елемент даних і має свій номер (насправді запис в кеші містить декілька елементів даних).

Між номерами рядків кеш-пам'яті і адресами оперативної пам'яті встановлюється відповідність «один до багатьох»: одному номеру рядка відповідає декілька адрес оперативної пам'яті.

В якості відображаючої функції може використовуватися просте виділення декількох розрядів з адреси оперативної пам'яті, які інтерпретуються як номер рядка кеш-пам'яті (таке відображення називається прямим). Наприклад, нехай в кеш-пам'яті може зберігатися 1024 записи. Тоді будь-яка адреса оперативної пам'яті може бути відображена на адресу кеш-пам'яті простим відділенням 10 двійкових розрядів (рис. 4.7).

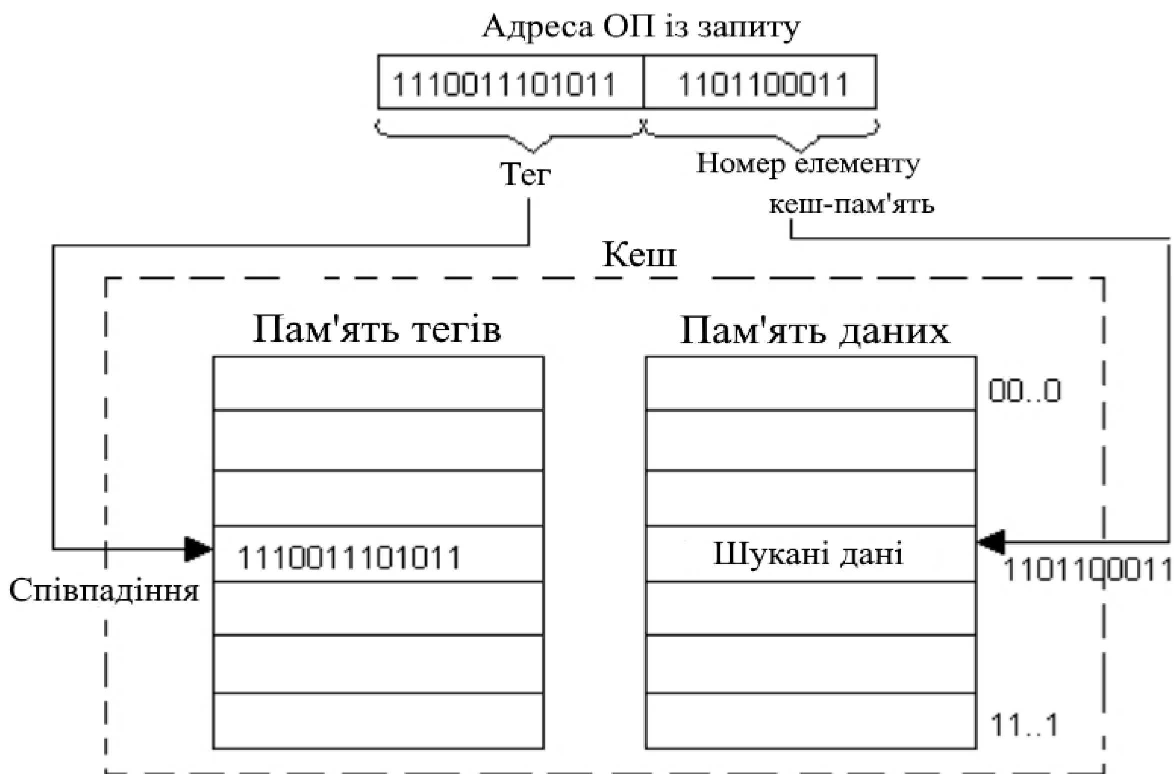


Рисунок 4.7 – Пряме відображення

При пошуку даних в кеші використовується швидкий прямий доступ до запису за номером рядка, отриманого шляхом обробки адреси оперативної пам'яті із запиту. Проте оскільки в знайденому рядку можуть знаходитися дані з будь-якого елементу оперативної пам'яті, молодші розряди адреси якої

співпадають з номером рядка, необхідно виконати додаткову перевірку. Для цих цілей кожен рядок кеш-пам'яті доповнюється тегом, що містить старшу частину адреси даних в оперативній пам'яті. При співпаданні тега з відповідною частиною адреси із запиту, констатується кеш-попадання.

Якщо ж стався кеш-промах, то дані прочитуються з оперативної пам'яті і копіюються в кеш. Якщо рядок кеш-пам'яті, в яку має бути скопійований елемент даних з оперативної пам'яті, містить інші дані, то останні витісняються з кеша. Відмітимо, що процес заміщення даних в кеш-пам'яті на основі прямого відображення істотно відрізняється від процесу заміщення даних в кеш-пам'яті з випадковим відображенням. По-перше, витіснення даних відбувається не лише у разі відсутності вільного місця в кеші, по-друге, ніякого вибору даних на заміщення не існує.

У багатьох сучасних процесорах кеш-пам'ять будується на основі поєднання цих двох підходів, що дозволяє знайти компроміс між порівняно низькою вартістю кеша з прямим відображенням і інтелектуальністю алгоритмів заміщення в кеші з випадковим відображенням (рис. 4.8).

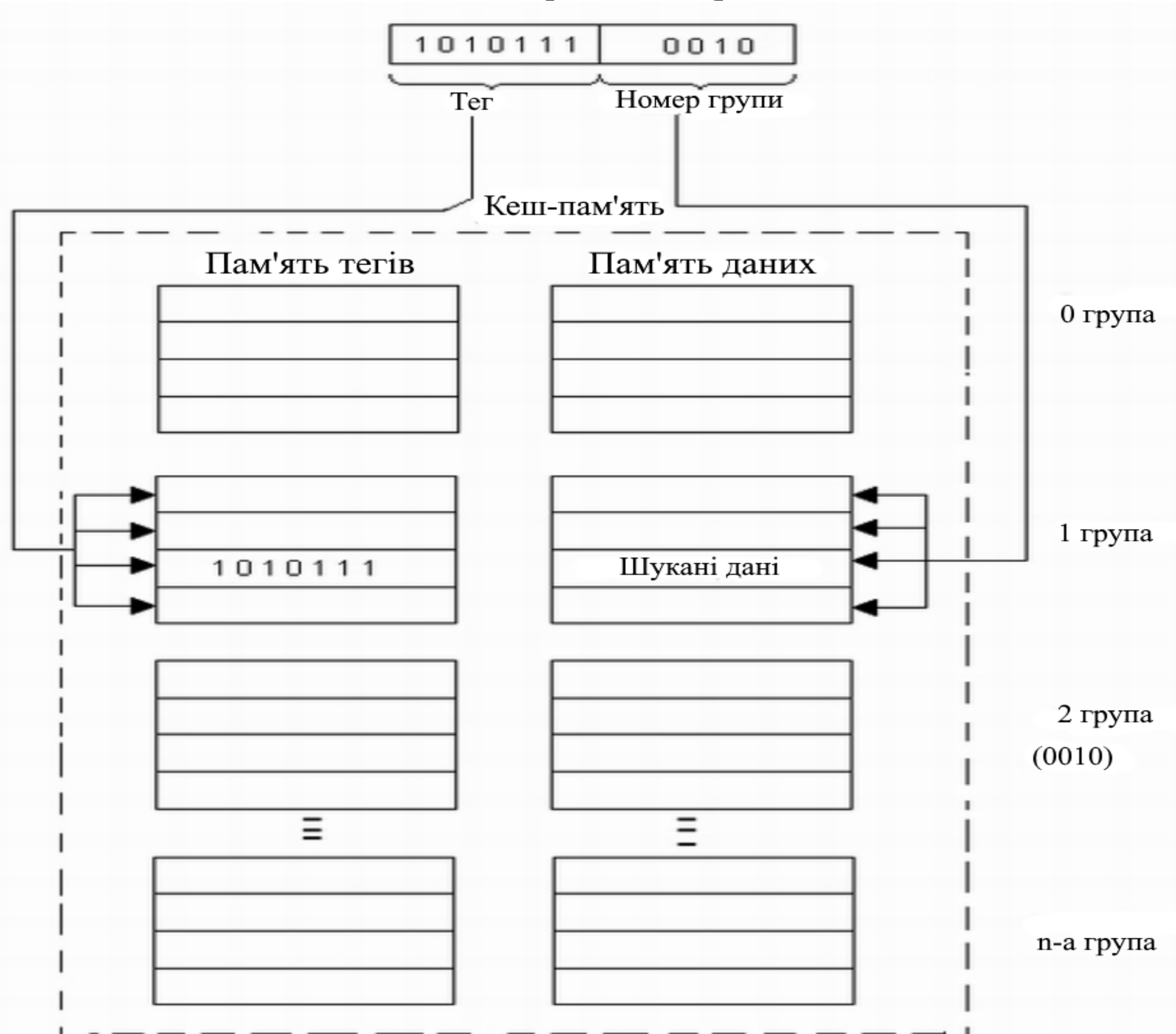


Рисунок 4.8 – Комбінування прямого і випадкового відображення

При змішаному підході довільна адреса оперативної пам'яті відображається не на одну адресу кеш-пам'яті (як це характерно для прямого відображення), і не на будь-яку адресу кеш-пам'яті (як це робиться при випадковому відображенні), а на деяку групу адрес. Усі групи пронумеровані. Пошук в кеші здійснюється спочатку за номером групи, отриманим з адреси оперативної пам'яті із запиту, а потім в межах групи шляхом асоціативного перегляду усіх записів в групі на предмет співпадання старших частин адрес оперативної пам'яті.

При промаху дані копіюються за будь-якою вільною адресою з однозначно заданої групи. Якщо вільних адрес в групі немає, то виконується витіснення даних. Оскільки кандидатів на вивантаження декілька – усі записи з цієї групи – алгоритм заміщення може врахувати інтенсивність звернень до даних і тим самим підвищити ймовірність попадань в майбутньому. Таким чином, у цьому способі комбінується пряме відображення на групу і випадкове відображення в межах групи.

4.6 Засоби підтримки механізмів віртуальної пам'яті

Засоби підтримки механізмів віртуальної пам'яті дозволяють відображати віртуальний адресний простір на фізичну пам'ять. Як приклад розглянемо механізм віртуальної пам'яті процесора Pentium.

Засоби підтримки механізмів віртуальної пам'яті в процесорі Pentium дозволяють відображати віртуальний адресний простір на фізичну пам'ять розміром максимум в 4 Гб. Цей максимум визначається використанням 32-розрядних адрес при роботі з оперативною пам'яттю.

Процесор може підтримувати як сегментну модель розподілу пам'яті, так і сегментно-сторінкову. Засоби сегментації утворюють верхній рівень засобів управління віртуальною пам'яттю процесора Pentium, а засоби сторінкової організації – нижній рівень. Це означає, що сегментні засоби працюють завжди, а засоби сторінкової організації можуть бути як включені, так і вимкнені шляхом установки однобітової ознаки PE (Paging Enable) в регістрі CRO процесора. Залежно від того, чи включені засоби сторінкової організації, змінюється сенс процедури перетворення адрес, яка виконується засобами сегментації. Спочатку розглянемо випадок роботи засобів сегментації при відключеному механізмі управління сторінками.

4.6.1 Віртуальний адресний простір

При роботі процесора Pentium у сегментному режимі в розпорядженні програміста є віртуальний адресний простір, що представляється сукупністю сегментів.

Кожен сегмент віртуальної пам'яті процесу має опис, який називається ***дескриптором сегменту***. Дескриптор сегменту має розмір 8 байт і містить усі характеристики сегменту, необхідні для перевірки правильності доступу до нього і знаходження його у фізичному адресному просторі (рис. 4.9).

Структура дескриптора, яка підтримується в процесорі Pentium, склалася історично. Багато що в ній пов'язане із забезпеченням сумісності з попередніми процесорами сімейства x86. Саме цим пояснюється те, що базова адреса сегменту представлена в дескрипторі у вигляді трьох частин, а розмір сегменту займає два поля.

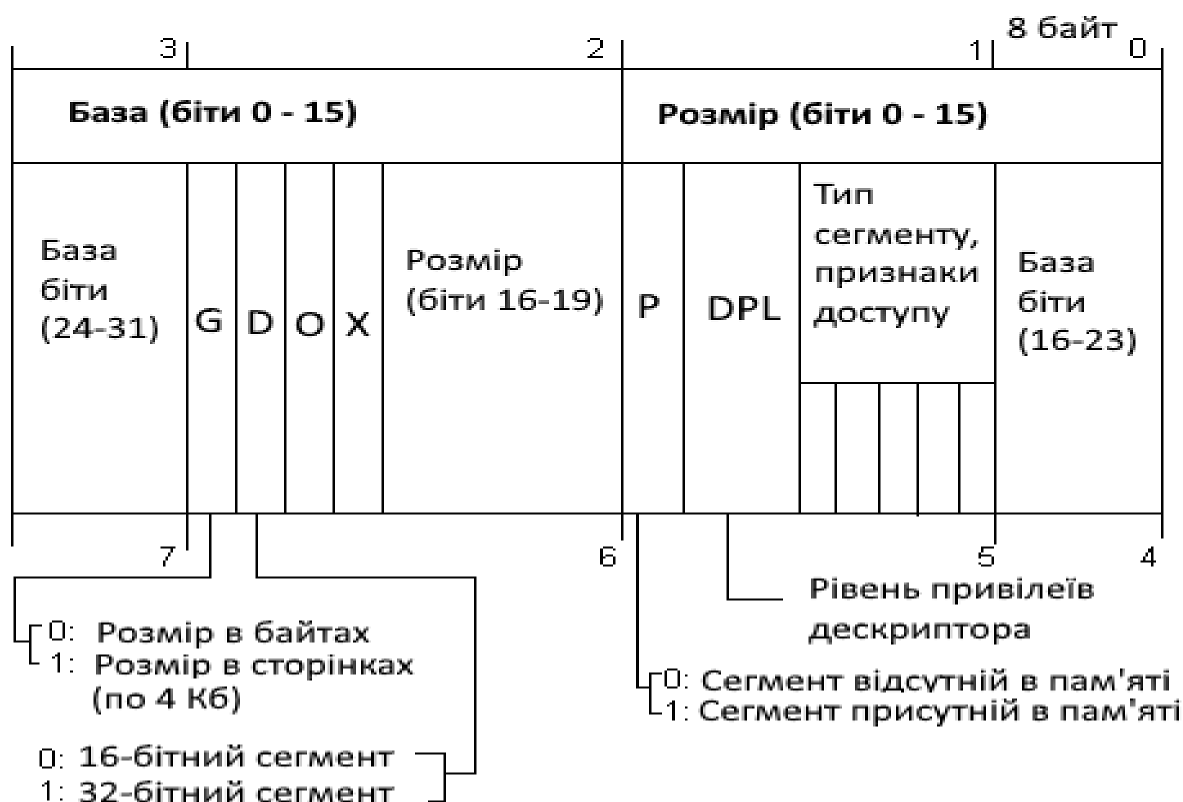


Рисунок 4.9 – Формат дескриптора сегмента даних або коду

Нижче перераховані основні поля дескриптора.

База – базова адреса сегменту (32 біти = 16+8+8), розділена на три частини із-за сумісності з i286, в якому це поле має тільки 24 біта.

Розмір – розмір сегменту (20 біт = 16+4), рознесений на дві частини.

G (Granularity) – одиниця виміру розміру сегменту, один біт. Якщо G=0, то розмір заданий в байтах і тоді сегмент не може бути більше 64 Кб, якщо G=1, то розмір сегменту вимірюється в сторінках по 4 Кб.

Байт доступу (5-й байт дескриптора) містить інформацію, яка використовується для ухвалення рішення про можливість або неможливість звернення до цього сегменту. Біт P (Present) визначає, чи знаходиться відповідний сегмент в даний момент в пам'яті (P=1) або він вивантажений на диск (P=0). Поле **DPL** (Descriptor Privilege Level, 2 біта) містить дані про рівень привілеїв, необхідний для доступу до сегменту (0-3). Інші 5 бітів байта доступу залежать від типу сегменту і визначають спосіб, яким можна використати цей сегмент (тобто читати, писати, виконувати). Відрізняються три основні типи сегментів: сегмент даних; кодовий сегмент; системний сегмент (GDT, TSS тощо).

Дескриптори сегментів об'єднуються в таблиці. Процесор Pentium для управління пам'яттю підтримує два типи таблиць дескрипторів сегментів:

- глобальну таблицю дескрипторів (Global Descriptor Table, GDT), яка призначена для опису сегментів операційної системи і загальних сегментів для усіх прикладних процесів, наприклад сегментів міжпроцесної взаємодії;
- локальну таблицю дескрипторів (Local Descriptor Table, LDT), яка містить дескриптори сегментів окремого призначеного для користувача процесу.

Таблиця GDT одна, а таблиць LDT стільки, скільки в системі виконується задач (процесів). При цьому в кожен момент часу операційною системою і апаратними засобами процесора використовується тільки одна з таблиць LDT, а саме та, яка відповідає виконуваному в даний момент призначеному для користувача процесу. Таблиця GDT описує загальну частину віртуального адресного простору процесів, а LDT – індивідуальну частину для кожного процесу. Таблиці GDT і LDT розміщені в оперативній пам'яті у вигляді окремих сегментів. Сегменти LDT і GDT містять системні дані, тому їх дескриптори зберігаються в таблиці GDT. Таким чином, таблиця GDT разом із записами про інші сегменти містить запис про саму собе, а також про усі таблиці LDT.

У кожен момент часу в спеціальних регістрах GDTR і LDTR зберігається інформація про місце розташування і розміри глобальної таблиці GDT і активної таблиці LDT відповідно. Регістр GDTR містить 32-розрядну фізичну адресу початку сегменту GDT в пам'яті, а також 16-бітовий розмір цього сегменту (рис. 4.10). Регістр LDTR вказує на розташування сегменту LDT в оперативній пам'яті опосередковано – він містить індекс дескриптора в таблиці GDT, в якому міститься адреса таблиці LDT і її розмір.

Рисунок 4.10 – Формат регістра GDTR

Процес звертається до фізичної пам'яті за віртуальною адресою (***селектор, зміщення***). Селектор однозначно визначає віртуальний сегмент, до якого належить шукана адреса, тобто він може інтерпретуватися як номер сегменту, а зміщення, як це і виходить з його назви, фіксує положення шуканої адреси відносно початку сегменту. Зміщення задається в машинній інструкції, а селектор поміщається в один із сегментних регістрів процесора. Під зміщення відводиться 32 біти, що забезпечує максимальний розмір сегменту 4 Гб (2^{32}).

Селектор витягується з одного з шести 16-розрядних сегментних регістрів процесора (CS, SS, DS, ES, FS або GS) залежно від типу команди і стадії її виконання – вибірки коду команди або даних. Наприклад, при зверненні до пам'яті під час вибірки наступної команди використовується селектор з сегментного регістра коду CS, а при записі результатів в сегмент даних процесу селектори витягуються з сегментних регістрів даних DS або ES. Якщо ж дані записуються в стек по команді PUSH, то механізм віртуальної пам'яті витягає селектор з сегментного регістра стека SS.

Селектор складається з трьох полів (рис. 4.11):

- індексу, який задає послідовний номер дескриптора в таблиці GDT або LDT (13 біт);
- покажчика типу використовуваної таблиці дескрипторів: GDT або LDT (1 біт);
- необхідного рівня привілеїв – RPL (2 біти), це поле використовується механізмом захисту даних (0 – ядро, 1 – системні вузли, 2 – бібліотеки спільного доступу, 3 – програми користувача). Рівні привілейованості забороняють виконуваному коду звернутися до нижчого рівня.

Віртуальний адресний простір процесу складається з усіх сегментів, описаних в загальній для усіх процесів таблиці GDT, і сегментів, описаних в його власній таблиці LDT. Розрядність (13) поля індексу визначає максимальне число глобальних і локальних сегментів процесу – по 8 Кб (2^{13}) сегментів кожного типу ($2*2^{13}$), всього 16 Кб сегментів.

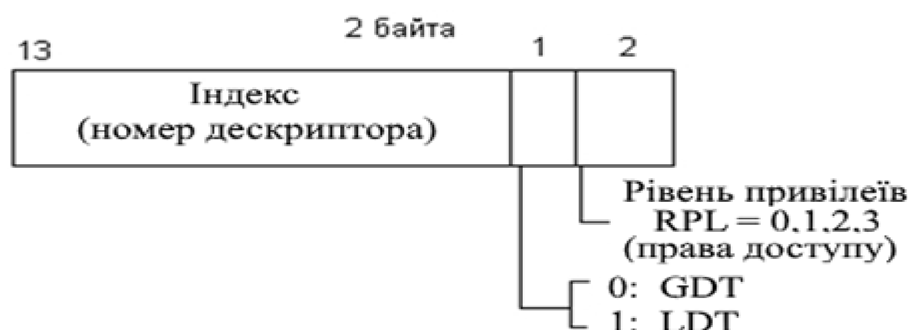


Рисунок 4.11 – Формат селектора в Pentium

З урахуванням максимального розміру сегменту – 4 Гб – кожен процес при чисто сегментній організації віртуальної пам'яті (без включення сторінкового механізму) може працювати у віртуалъному адресному просторі в 64 Тбайт ($16Кб*2^{32} = 2^{14}*2^{32} = 2^{46} = 2^{6}*2^{40}$).

Кожен дескриптор в таблицях GDT і LDT має розмір 8 байт, тому максимальний розмір кожної з цих таблиць – 64 Кб (8 байт * 8 Кб дескрипторів). З приведеного опису видно, що процесор Pentium забезпечує підтримку роботи ОС у двох відношеннях:

- підтримує роботу віртуальної пам'яті за рахунок швидкого апаратного способу перетворення віртуальної адреси у фізичну;
- забезпечує захист даних і кодів різних додатків.

4.6.2 Перетворення адрес

Тепер простежимо, яким чином віртуальний простір в 64 Тбайт відображається на фізичний простір розміром в 4 Гб. Механізм відображення перетворює віртуальну адресу, яка представлена селектором, що знаходиться в одному з сегментних регістрів, і зміщенням, витягнутим з відповідного поля машинної інструкції, в лінійну фізичну адресу.

Розглянемо спочатку випадок, коли віртуальна адреса належить до одного з сегментів, дескриптори яких містяться в таблиці GDT (рис. 4.12).

1. Значення селектора вказує механізму перетворення адрес, що віртуальна адреса належить до сегменту, що описується в таблиці GDT. Місцезнаходження таблиці GDT система визначає з регістра GDTR, в якому зберігається повна 32-бітова базова фізична адреса таблиці. Процесор складає базову адресу таблиці, взяту з регістра GDTR, із зрушеним на 3 розряди вліво (множення на 8 відповідно до числа байтів в одному дескрипторі сегменту) значенням поля індексу з селектора. Результатом є фізична адреса дескриптора сегменту, до якого належить задана віртуальна адреса.

2. За обчисленою адресою процесор витягає з пам'яті дескриптор потрібного сегменту.

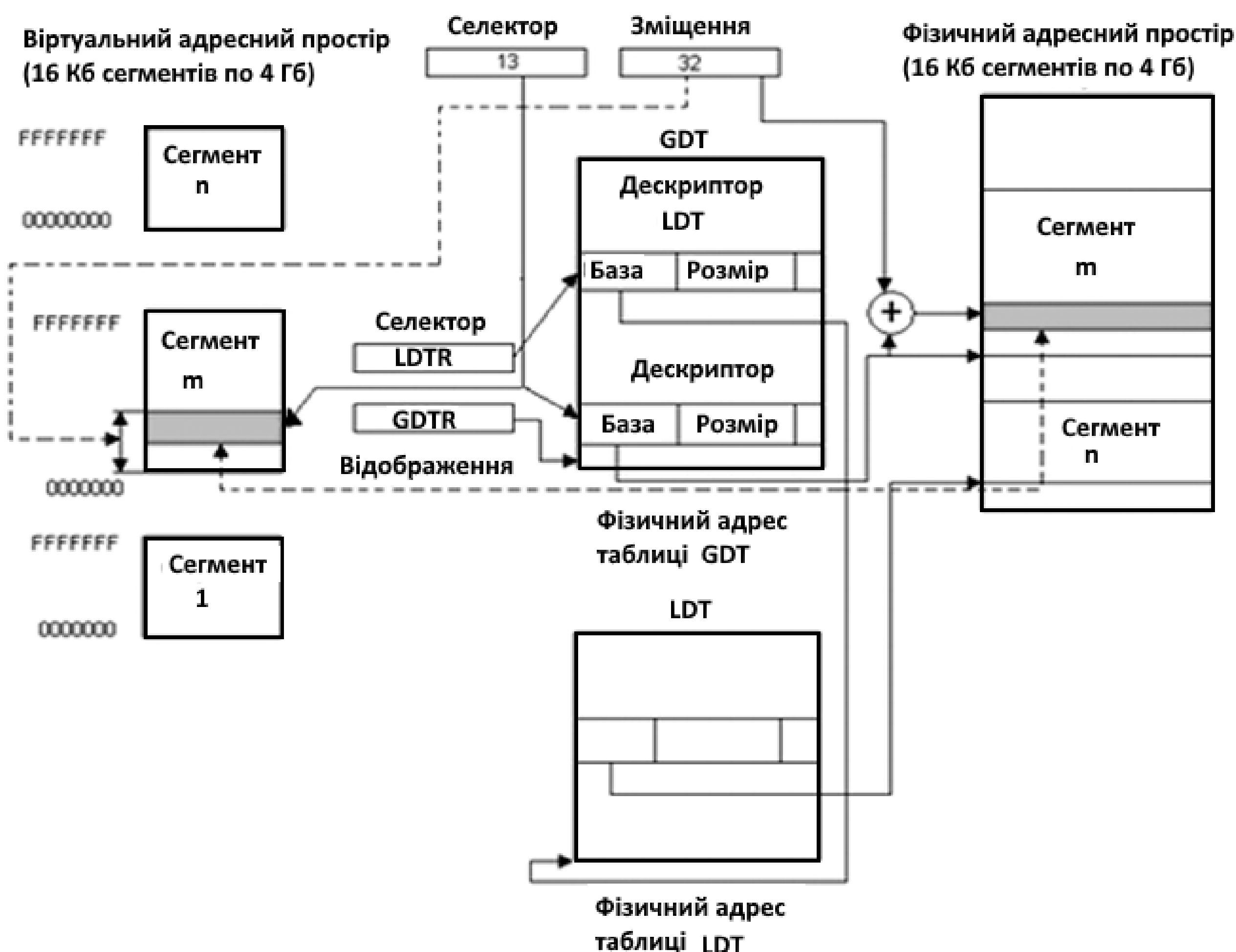


Рисунок 4.12 – Механізм перетворення віртуальної адреси у фізичну при роботі процесора в сегментному режимі

3. Здійснюється перевірка можливості виконання заданої операції доступу за заданою віртуальною адресою:

- спочатку процесор визначає правильність адреси, порівнюючи зміщення, задане у віртуальній адресі, з розміром сегменту, витягнутим з регістра LDTR (у разі виходу адреси за межі сегменту відбувається переривання);
- потім процесор перевіряє права доступу процесу до сегменту пам'яті;
- далі перевіряється наявність сегменту у фізичній пам'яті (якщо біт P = 0, тобто сегмент відсутній у фізичній пам'яті, то відбувається переривання).

4. Якщо всі три умови виконані, то доступ за заданою віртуальною адресою дозволений. Виконується перетворення віртуальної адреси у фізичну шляхом складання базової адреси сегменту, витягнутої з дескриптора, і зміщення, заданого в інструкції. Виконується задана операція над елементом фізичної пам'яті за цією адресою.

У разі, коли селектор у віртуальній адресі вказує не на таблицю GDT, а на таблицю LDT, процедура обчислення фізичної адреси дещо ускладнюється. Це пов'язано з тим, що регістр LDTR на відміну від GDTR вказує на розміщення таблиці сегментів не прямо, а побічно. У LDTR міститься індекс дескриптора сегмента LDT. Тому в процедурі перетворення адрес з'являється додатковий етап – визначення базової адреси таблиці LDT. На підставі базової адреси таблиці GDT, узятої з регістра GDTR, і селектора, взятого з регістра LDTR, обчислюється зміщення в таблиці GDT, яке і є адресою дескриптора сегменту LDT.

З дескриптора витягається базова адреса таблиці LDT, і з цієї миті робота механізму відображення повністю аналогічна описаному вище перетворенню віртуальної адреси за допомогою таблиці GDT. Тобто, на підставі базової адреси таблиці LDT і селектора процесу, заданого в одному з сегментних регістрів, обчислюється зміщення в таблиці LDT і визначається базова адреса дескриптора шуканого сегменту. З цього дескриптора витягається базова адреса сегменту, який складається зі зміщенням з віртуальної адреси, що і дає в результаті шукану фізичну адресу.

Таким чином, для використання сегментного механізму процесора Pentium операційній системі необхідно сформувати таблиці GDT і LDT, завантажити їх в пам'ять (спершу досить завантажити тільки таблицю GDT), завантажити покажчики на ці таблиці в регістри GDTR і LDTR і вимкнути сторінкову підтримку.

Операційна система може відмовитися від використання засобів сегментації процесора Pentium. У такому разі їй достатньо призначити кожному процесу тільки по одному сегменту і занести у відповідні таблиці LDT по одному дескриптору. Віртуальний адресний простір процесу складається з одного сегменту завдовжки максимум в 4 Гб. Оскільки вивантаження процесів на диск здійснюватиметься повністю, віртуальна пам'ять вироджується в окремому випадку у свопінг.

4.6.3 Сегментно-сторінковий механізм

Включення сторінкового механізму відбувається, якщо в регістрі управління CR0 найстарший біт PG встановлений в одиницю. При включеній системі управління сторінками паралельно продовжує працювати і описаний вище сегментний механізм, проте, як буде показано нижче, сенс його роботи міняється.

Віртуальний адресний простір процесу при сегментно-сторінковому режимі роботи процесора обмежується розміром 4 Гб, і при 4К-байтній сторінці, сегмент може містити 1 млн сторінок (4Гб/4Кб). У цьому просторі визначені віртуальні сегменти процесу (рис. 4.13). Оскільки тепер усі віртуальні сегменти

розділяють один віртуальний адресний простір, то можливе їх накладення, оскільки процесор не контролює такі ситуації, залишаючи цю проблему операційній системі.

Для реалізації механізму управління сторінками як фізичний, так і віртуальний адресні простори розбиті на сторінки розміром 4 Кб. Починаючи з моделі Pentium в процесорах Intel існує можливість використання сторінок і по 4 Мб, але подальший виклад орієнтується на традиційний розмір сторінки в 4 Кб. Всього у віртуальному адресному просторі в сегментно-сторінковому режимі налічується 1 Мб (2^{20} = 4Гб/4Кб) сторінок. Незважаючи на наявність декількох віртуальних сегментів, увесь віртуальний адресний простір процесу має загальне розбиття на сторінки, так що **нумерація віртуальних сторінок наскрізна**.

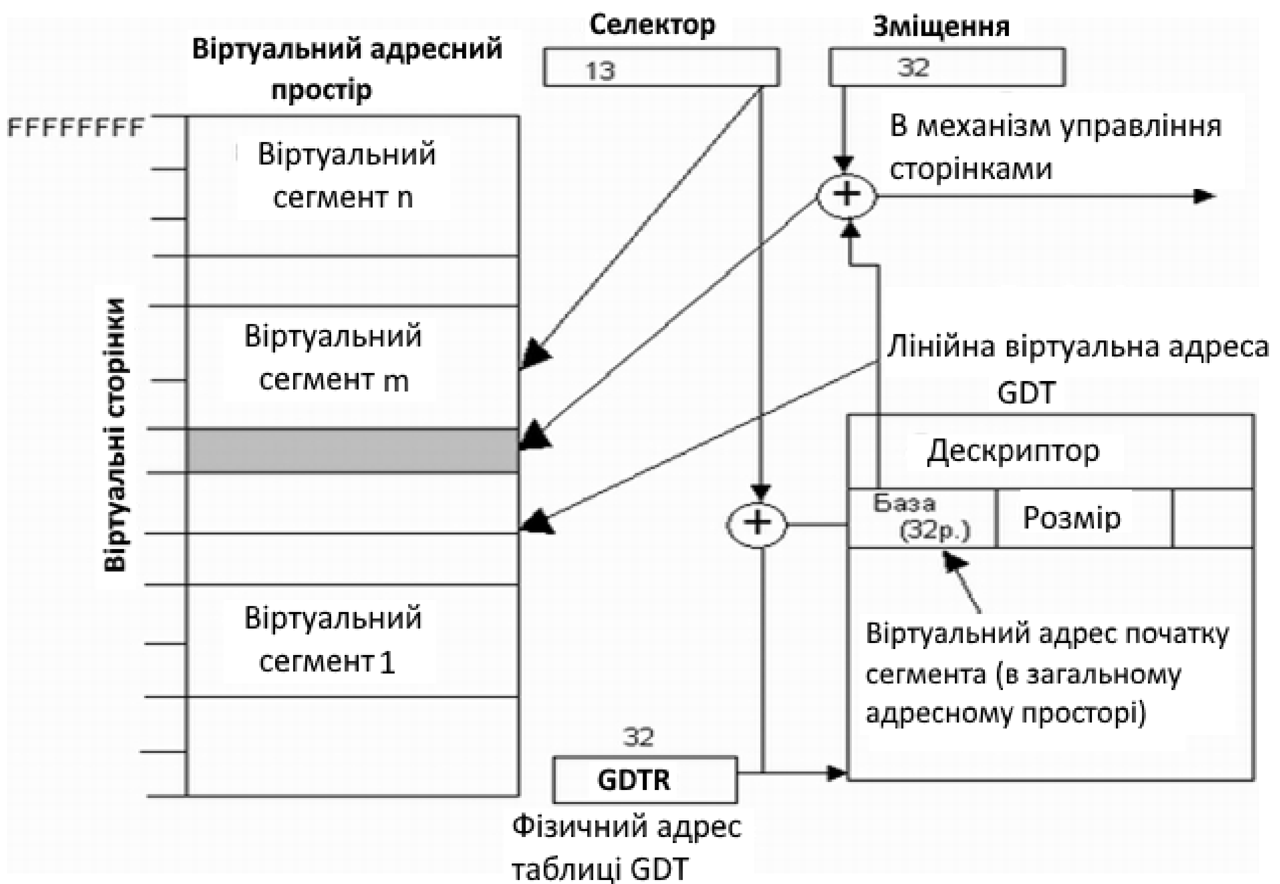


Рисунок 4.13 – Робота механізму в сегментно-сторінковому режимі

Віртуальна адреса як і раніше представляє собою пару: селектор, який визначає номер віртуального сегменту, і зміщення усередині цього сегменту. Перетворення віртуальної адреси виконується в два етапи: спочатку працює сегментний механізм, а потім результат його роботи поступає на вхід сторінкового механізму, який і обчислює шукану фізичну адресу.

Робота сегментного механізму в даному випадку багато в чому повторює його роботу при відключеному сторінковому механізмі. На підставі значення індексу в селекторі вибирається потрібний дескриптор з таблиці GDT або LDT. З дескриптора витягається базова адреса сегменту і складається зі зміщенням.

Дескриптори і таблиці мають ту ж структуру. Проте є і принципова відмінність, полягає в інтерпретації утримуваного поля базової адреси в дескрипторах сегментів. Якщо раніше дескриптор сегменту містив базову адресу сегменту у фізичній пам'яті і при складанні цієї адреси зі зміщенням з віртуальної адреси виходила фізична адреса, то тепер дескриптор містить базову адресу сегменту у віртуальному адресному просторі, і в результаті його складання зі зміщенням виходить ***лінійна віртуальна адреса.***

Результуюча лінійна 32-розрядна віртуальна адреса передається сторінковому механізму для подальшого перетворення. Так як розмір сторінки дорівнює 4 Кб (2^{12}), то в адресі можна легко виділити номер віртуальної сторінки (старші 20 розрядів) і зміщення в сторінці (молодші 12 розрядів).

Для відображення віртуальної сторінки у фізичну досить побудувати таблицю сторінок, кожен елемент якої – дескриптор віртуальної сторінки – містив би номер фізичної сторінки, яка відповідала б їй, і її атрибутам (рис. 4.14). Двадцять розрядів, в яких знаходиться номер сторінки, можуть інтерпретуватися і як базова адреса сторінки в пам'яті, яку необхідно доповнити 12 нулями, оскільки молодші 12 розрядів базової адреси сторінки завжди дорівнюють нулю. Окрім номера сторінки, дескриптор сторінки містить також поля, близькі за змістом відповідним полям дескриптора сегменту:

- P – біт присутності сторінки у фізичній пам'яті;
- W – біт дозволу запису в сторінку;
- U – біт користувач/супервізор;
- A – ознака доступу, що мав місце до сторінки;
- D – ознака модифікації вмісту сторінки;
- PWT і PCD – управляють механізмом кешування сторінок (введені починаючи з процесора i486);
- AVL – резерв для потреб операційної системи (AVaiLable for use).

20	3	2	1	1	1	1	1	1	1
№ сторінки	AVL	0	D	A	PCD	PWT	U	W	P

Рисунок 4.14 – Формат дескриптора сторінки

При невеликому розмірі сторінки процесора Pentium порівняно з розмірами адресних просторів таблиця сторінок повинна займати в пам'яті значне місце – 4 байт * 1 Мб = 4 Мб. Це надто багато для нинішніх моделей персональних комп'ютерів, тому в процесорі Pentium використовується ділення усієї таблиці сторінок на частини – ***розділи*** по 1024 дескриптори. Розмір розділу вибраний так, щоб один розділ займав одну фізичну сторінку(1024 * 4 байт – 4 Кб). Таким чином таблиця сторінок ділиться на 1024 розділи.

Щоб постійно не зберігати в пам'яті всі розділи, створюється ***таблиця розділів (каталог сторінок***), що складається з дескрипторів розділів, які мають таку ж структуру, що і дескриптори сторінок. Максимальний розмір таблиці розділів складає 4 Кб, тобто одна сторінка. Віртуальні сторінки, що містять

розділи, як і усі інші сторінки, можуть вивантажуватися на диск. Віртуальна сторінка, що зберігає таблицю розділів, завжди знаходиться у фізичній пам'яті, і номер її фізичної сторінки вказується в спеціальному управляючому регістрі CR3 процесора. Перетворення лінійної віртуальної адреси у фізичну відбувається таким чином (рис. 4.15).

Поле номеру віртуальної сторінки (старші 20 розрядів) ділиться на дві рівні частини по 10 розрядів – поле номера розділу і поле номера сторінки в розділі. На підставі заданого в регістрі CR3 номера фізичної сторінки, що зберігає таблицю розділів, і зміщення в цій сторінці, що задається полем номеру розділу, процесор знаходить дескриптор віртуальної сторінки розділу.

Відповідно до атрибутів цього дескриптора визначаються права доступу до сторінки, а також наявність її у фізичній пам'яті. Якщо сторінки немає в оперативній пам'яті, то відбувається переривання, в результаті якого операційна система повинна виконати завантаження необхідної сторінки в пам'ять.

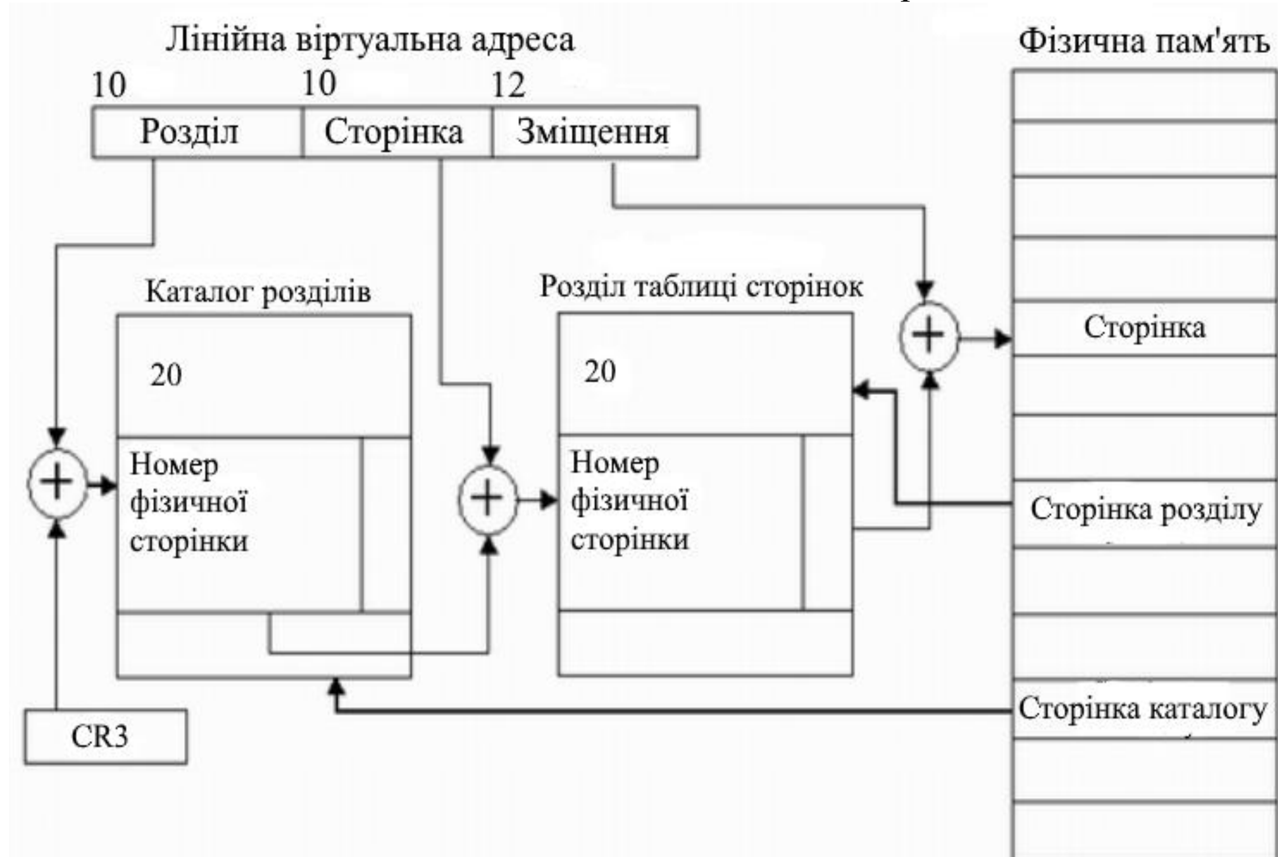


Рисунок 4.15 – Перетворення лінійної віртуальної адреси у фізичну адресу

Після того як сторінка, що містить потрібний розділ, завантажена, з неї витягається дескриптор сторінки даних, номер якої вказаний в лінійній віртуальній адресі. І нарешті, на підставі базової адреси сторінки, отриманої з дескриптора, і зміщення, заданого в лінійній віртуальній адресі, обчислюється шукана фізична адреса. Таким чином, при доступі до сторінки в процесорі використовується дворівнева схема адресації сторінок, яка хоч і уповільнює перетворення, але дозволяє використати сторінковий механізм для таблиці сторінок, що істотно зменшує об'єм фізичної пам'яті, потрібної для її зберігання.

Для прискорення перетворення адрес у блоці управління сторінками використовується асоціативна пам'ять, в якій знаходиться 32 дескриптори активно використовуваних сторінок, що дозволяє за номером віртуальної сторінки швидко знайти номер фізичної сторінки без звернення до таблиць розділів і сторінок.

4.7 Кешування в процесорі Pentium

У процесорі Pentium кешування використовується у випадках, описаних нижче.

Кешування дескрипторів сегментів в прихованих регістрах. Для кожного сегментного регістра в процесорі є так званий прихований регістр дескриптора. У прихований регістр при завантаженні сегментного регістра поміщається інформація з дескриптора, на який вказує цей сегментний регістр. Інформація з дескриптора сегменту використовується для перетворення віртуальної адреси у фізичну при чисто сегментній організації пам'яті або для отримання лінійної віртуальної адреси при сторінковому механізмі. Доступ до прихованого регістра виконується швидше, ніж пошук і витягання інформації з таблиці сторінок, що знаходиться в оперативній пам'яті. Тому, якщо чергове звернення належатиме до одного з сегментів, дескриптор якого ще зберігається в прихованому регістрі (а ймовірність цього велика), то перетворення адрес буде виконане швидше. Тим самим приховані регістри грають роль кеша таблиці дескрипторів і прискорюють роботу процесора.

Кешування пар номерів віртуальних і фізичних сторінок у буфері асоціативної трансляції TLB (Translation Lookaside Buffer) дозволяє прискорювати перетворення віртуальних адрес у фізичні при сегментно-сторінковій організації пам'яті. TLB є асоціативною пам'яттю невеликого об'єму, призначеною для зберігання інтенсивно використовуваних дескрипторів сторінок. У процесорі Pentium є окремі TLB для інструкцій і даних.

Кешування даних і інструкцій в кеш-пам'яті першого рівня. Ця пам'ять ще називається також внутрішньою кеш-пам'яттю, оскільки вона розміщена безпосередньо на кристалі мікропроцесора і має об'єм 16/32 Кб. У процесорі Pentium кеш першого рівня розділений на пам'ять для зберігання даних і пам'ять для зберігання інструкцій.

Кешування даних і інструкцій в кеш-пам'яті другого рівня. Ця пам'ять називається також зовнішньою кеш-пам'яттю, оскільки вона встановлюється у вигляді окремої мікросхеми на системній платі. Кеш-пам'ять другого рівня є загальною для даних і інструкцій і має об'єм 256/512 Кб. Пошук в кеші другого рівня виконується у разі, коли констатується промах в кеші першого рівня. Для узгодження даних в кеші другого рівня може використовуватися як наскрізний, так і зворотний запис.

Розглянемо детальніше принципи роботи буфера асоціативної трансляції і кеша першого рівня.

4.7.1 Буфер асоціативної трансляції

У принципі кожна віртуальна адреса викликає звернення до двох фізичних адрес: одне для вибірки відповідного запису з таблиці сторінок, і ще одне – для звернення до адресних даних. А в разі використання дворівневих таблиць сторінок потрібні три операції доступу: до каталогу (розділу) сторінок, до таблиці сторінок і безпосередньо за фізичною адресою. Отже, проста схема віртуальної пам'яті, по суті, подвоює звернення до пам'яті.

Проблема прискорення пошуку вирішується на рівні архітектури комп'ютера. Відповідно до властивості локальності більшість програм впродовж деякого проміжку часу звертаються до невеликої кількості сторінок, тому активно використовується тільки невелика частина таблиці сторінок. Комп'ютер забезпечується апаратним пристроєм для відображення віртуальних сторінок у фізичні без звернення до таблиці сторінок, який має невелику, швидку кеш-пам'ять, що зберігає необхідну на даний момент частину таблиці сторінок.

Для вирішення цієї проблеми більшість схем віртуальної пам'яті використовують спеціальний високошвидкісний кеш для записів таблиць сторінок, який називають ***буфером швидкого перетворення адреси,*** або ***буфером пошуку трансляції*** (translation lookaside buffer – TLB, або ***буфер асоціативної трансляції***, іноді і ***асоціативною пам'яттю*** (рис. 4.16)).

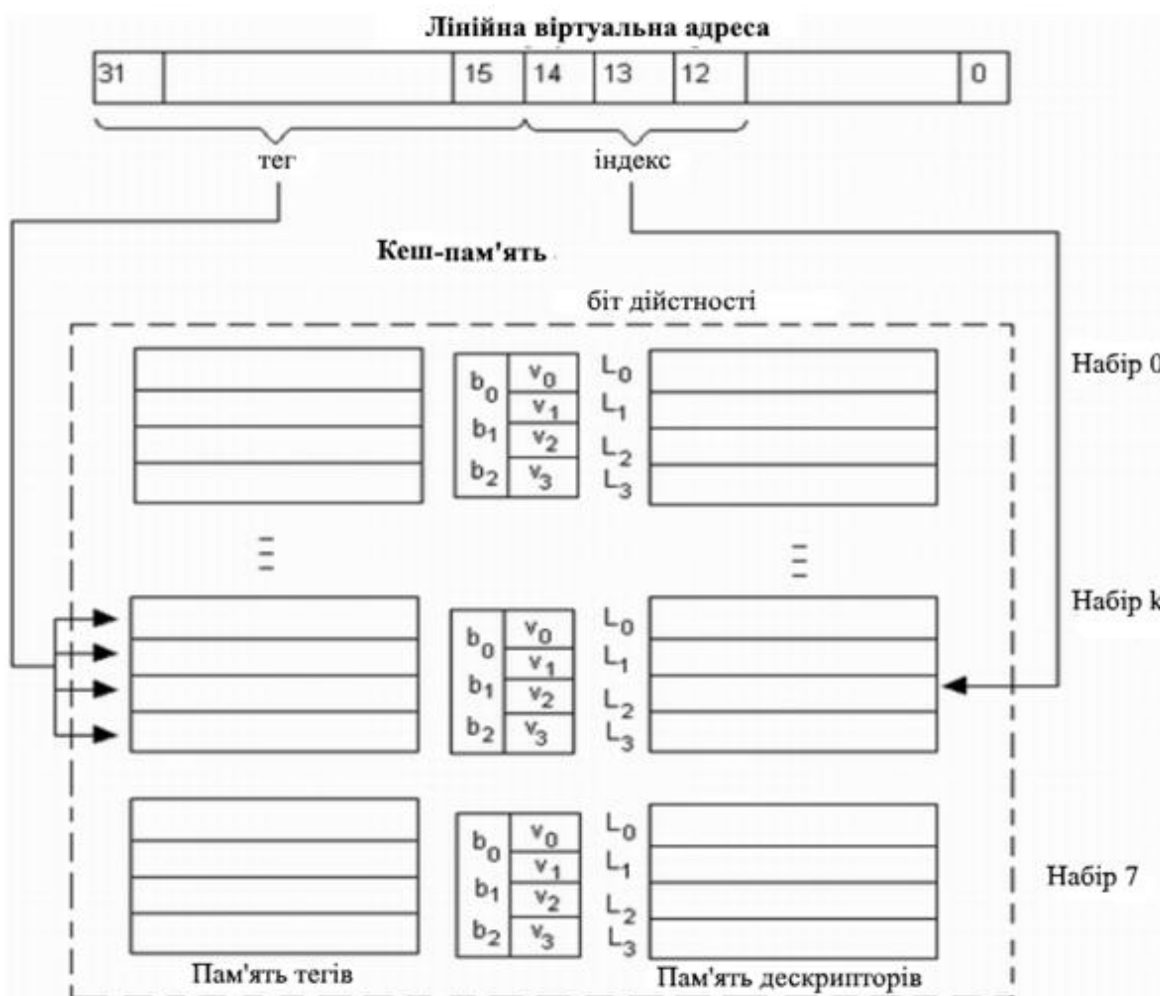


Рисунок 4.16 – Буфер асоціативної трансляції

Грунтуючись на правилі «*дев'яносто до десяти*» (правилі локалізації), яке стверджує, що більшість програм схильна робити величезну кількість звернень до невеликої кількості сторінок, комп'ютер забезпечується невеликим апаратним пристроєм, що служить для відображення віртуальних адрес у фізичні без проходу таблицею сторінок. Цей пристрій знаходиться усередині диспетчера пам'яті і складається з декількох записів, від 8 до 4096. Так, в архітектурі Intel-32 таких елементів до Pentium-4 було 32 (що забезпечується 98% попадань в кеш), починаючи з Pentium-4 – 128 елементів.

У буфері TLB кешуються дескриптори сторінок з таблиці сторінок. Для зберігання дескриптора в кеші відводиться один рядок. Кожен рядок доповнений тегом, в якому міститься номер відповідної віртуальної сторінки. Рядки об'єднані по чотири в групи, що називаються наборами.

Таблиця TLB, яка використовується для перетворення адрес інструкцій, має 32 рядки і відповідно до 8 наборів. Номер набору називають індексом (index). Таким чином, шляхом кешування може бути отримана фізична адреса для доступу до 32 сторінок пам'яті, що містять інструкції.

Після того як механізмом сегментації отримана лінійна адреса, він має бути перетворений у фізичну адресу. Для цього передусім необхідно знайти дескриптор сторінки, до якої належить ця адреса, і витягнути з нього номер фізичної сторінки. Звичайна процедура передбачає звернення до таблиці розділів, а потім до таблиці сторінок. Проте фізична адреса може бути отримана набагато швидше завдяки тому, що в буфері TLB зберігаються копії дескрипторів найбільш інтенсивно використовуваних сторінок. Тому перед тим, як почати порівняно тривалу процедуру перетворення адрес, робиться спроба виявити потрібний дескриптор сторінки в швидкій асоціативній пам'яті TLB. Потім на підставі номера фізичної сторінки, отриманого з TLB, обчислюється фізична адреса.

При пошуку даних в TLB використовується лінійна віртуальна адреса. Розряди 12-14 використовуються як індекс набору. Далі перевіряються біти дійсності (**V**) усіх рядків вибраного набору. На початку роботи кеш-пам'яті біти дійсності усіх рядків скидаються в нуль. Біт дійсності набуває значення 1, коли у відповідному рядку міститься достовірна інформація, і скидається в нуль, коли рядок оголошується вільним, в результаті роботи алгоритму заміщення. Для усіх дійсних рядків виконується асоціативна процедура порівняння тегів із старшими розрядами (15-31 розряд) лінійної віртуальної адреси. Якщо сталося кеш-попадання, то номер фізичної сторінки швидко поступає в схему формування фізичної адреси.

Якщо стався промах і потрібного дескриптора в TLB немає, то запускається багатоетапна процедура перетворення адреси, що включає звернення до таблиць розділів і сторінок. Коли потрібний дескриптор відшукується в таблиці сторінок, він копіюється в TLB. Номер набору, в який записується кешований дескриптор, визначається трьома молодшими розрядами номера віртуальної сторінки (розряди 12-14 лінійної віртуальної адреси).

Проте оскільки в наборі є чотири рядки, необхідно визначити, в яку саме потрібно помістити кешовані дані. Дескриптор записується або в перший

вільний рядок, що попався, або, якщо усі рядки зайняті, в рядок, до якого найдовше не зверталися. Ознакою зайнятості рядка служить біт дійсності **V**, наявний у кожного рядка. Якщо **V=0**, значить, рядок вільний для запису в нього нового вмісту. Для визначення рядка, який не використовувався довше за всіх інших в цьому наборі, застосовується спрощений варіант алгоритму PseudoLRU (Pseudo Least Recently Used). Цей алгоритм грунтується на аналізі трьох біт: b0, b1, b2, що називаються бітами звернення. Бити звернення приписуються набору і встановлюються відповідно до алгоритму, наведеному на рис. 4.17. Тут L0, L1, L2, L3 означають послідовні рядки набору. На заміну вибирається один з таких рядків:

– L0, якщо b0=0 і b1=0;
– L1, якщо b0=0 і b1=1;
– L2, якщо b0=1 і b2=0;
– L3, якщо b0=1 і b2=1.

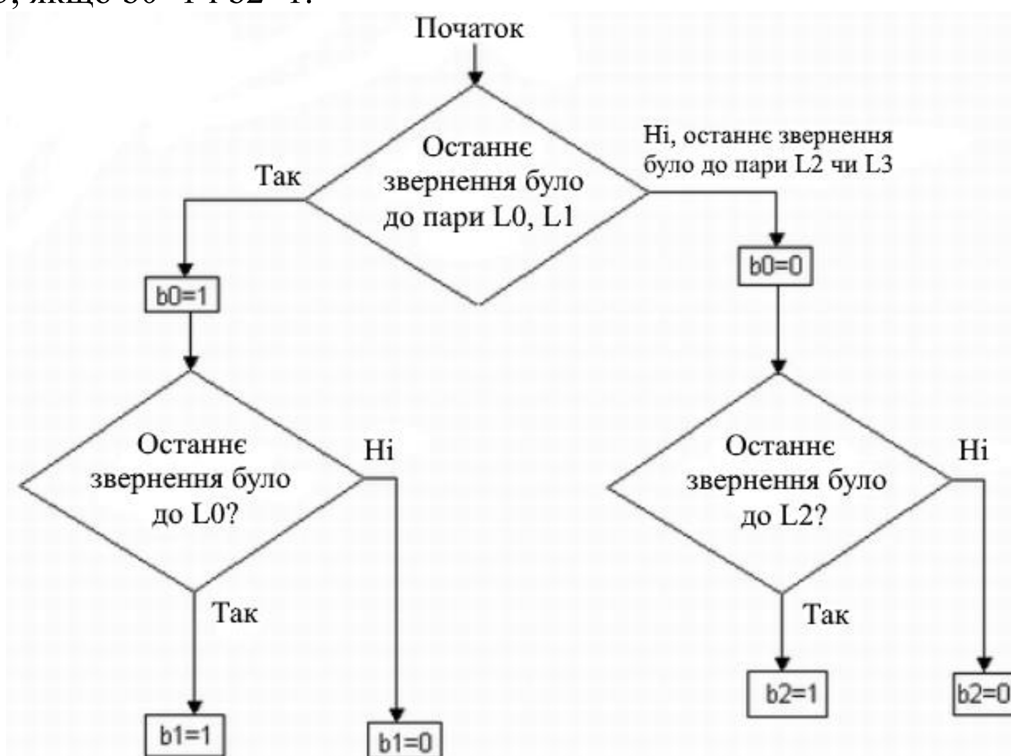


Рисунок 4.17 – Алгоритм установки бітів звернення

Можна легко показати, що ця процедура не завжди призводить до вибору дійсно довшого за всіх рядка, що не викликався. Нехай, наприклад, звернення до рядків виконувалися в наступній хронологічній послідовності: L0, L2, L3, L1, тобто найближче за часом звернення було до рядка L1, найдовше ж не було звернень до рядка L0. Бити звернення в даному випадку набудуть наступних значень. Оскільки останнє за часом звернення було до рядка з пари (L0, L1), значить, b0=1. А в парі (L2, L3) останнє звернення було до L3, отже, b2=0. Звідси, за правилом, наведеним вище, на заміну вибирається рядок L2, замість рядка L0, до якого насправді найдовше не було звернень.

Проте в більшості випадків цей алгоритм дає результат, співпадаючий з оптимальним. Наприклад, для послідовності L0, L3, LI, L2 біти звернення мають

значення b0=0, b1=0, звідси точне рішення – L0. Навіть у разі помилки (ймовірність якої складає 0,3) рішення, знайдені за алгоритмом PseudoLRU, близькі до оптимальних. Так, у першому прикладі замість рядка L0, що є правильним рішенням, алгоритм дав найближчий до нього за часом звернення рядок L2.

Незважаючи на те що алгоритм PseudoLRU дає в загальному випадку наближені рішення, він широко застосовується при кешуванні, оскільки є швидким і економічним, що надзвичайно важливо для кеш-пам'яті.

Таким чином, у буфері TLB процесора Pentium використовується комбінований спосіб відображення кешованих даних на кеш-пам'ять: пряме відображення дескрипторів на набори і випадкове відображення на рядки в межах набору.

Наявність TLB дозволяє в переважному числі випадків замінити порівняно довгу процедуру перетворення адрес, пов'язану з декількома зверненнями до оперативної пам'яті, швидким пошуком в асоціативній пам'яті.

4.7.2 Кеш першого рівня

Кеш першого рівня використовується на етапі обробки запиту до основної пам'яті за фізичною адресою. Робота кеш-пам'яті першого рівня має багато спільного з роботою буфера TLB. У TLB одиницею зберігання є дескриптор, а в кеші першого рівня – байт даних. Оновлення даних в кеші відбувається блоками по 16 байт. Таким чином, молодші 4 біти фізичної адреси байта можуть інтерпретуватися як зміщення в блоці, а старші розряди – як номер блоку (рис. 4.18).

Для зберігання блоків даних в кеші відводяться рядки, що також мають об'єм 16 байт. Рядки об'єднані в набори по чотири. При об'ємі кеша 16 Кб в нього входять 256 (2^8) наборів ($16*2^{10} = 2^4*2^{10}/2^4*2^2 = 2^8$).

При копіюванні даних в кеш номери блоків основної пам'яті прямо відображаються на номери наборів. Для цього в адресі основної пам'яті, що належить до одного з байтів, що входять у блок, значення 8 бітів, що знаходяться перед бітами зміщення, інтерпретується як номер набору в кеш-пам'яті. Інші старші біти адреси надалі використовуються як тег.

Так само як в TLB, вибір рядка в наборі здійснюється на основі аналізу бітів дійсності і бітів звернення за алгоритмом PseudoLRU. Блок даних заноситься в рядок кеш-пам'яті разом зі своїм тегом. Біт дійсності рядка встановлюється в 1.

При виникненні запиту на читання з основної пам'яті спочатку робиться спроба знайти дані в кеші. За індексом, витягнутим з адреси запиту, визначається набір, в якому можуть знаходитися шукані дані. Потім для рядків цього набору виконується асоціативний пошук. Старші розряди адреси із запиту порівнюються з тегами усіх рядків набору. Якщо для якого-небудь рядка фіксується співпадання, це означає, що сталося ***кеш-попадання***, і з відповідного рядка витягається байт, зміщення якого відносно початку рядка визначається чотирма молодшими розрядами з адреси запиту.

Для узгодження даних в кеші 1-го рівня використовується метод наскрізного запису, тобто при виникненні запиту на запис оновлюється не лише вміст відповідного елементу основної пам'яті, але і його копія в кеш-пам'яті.

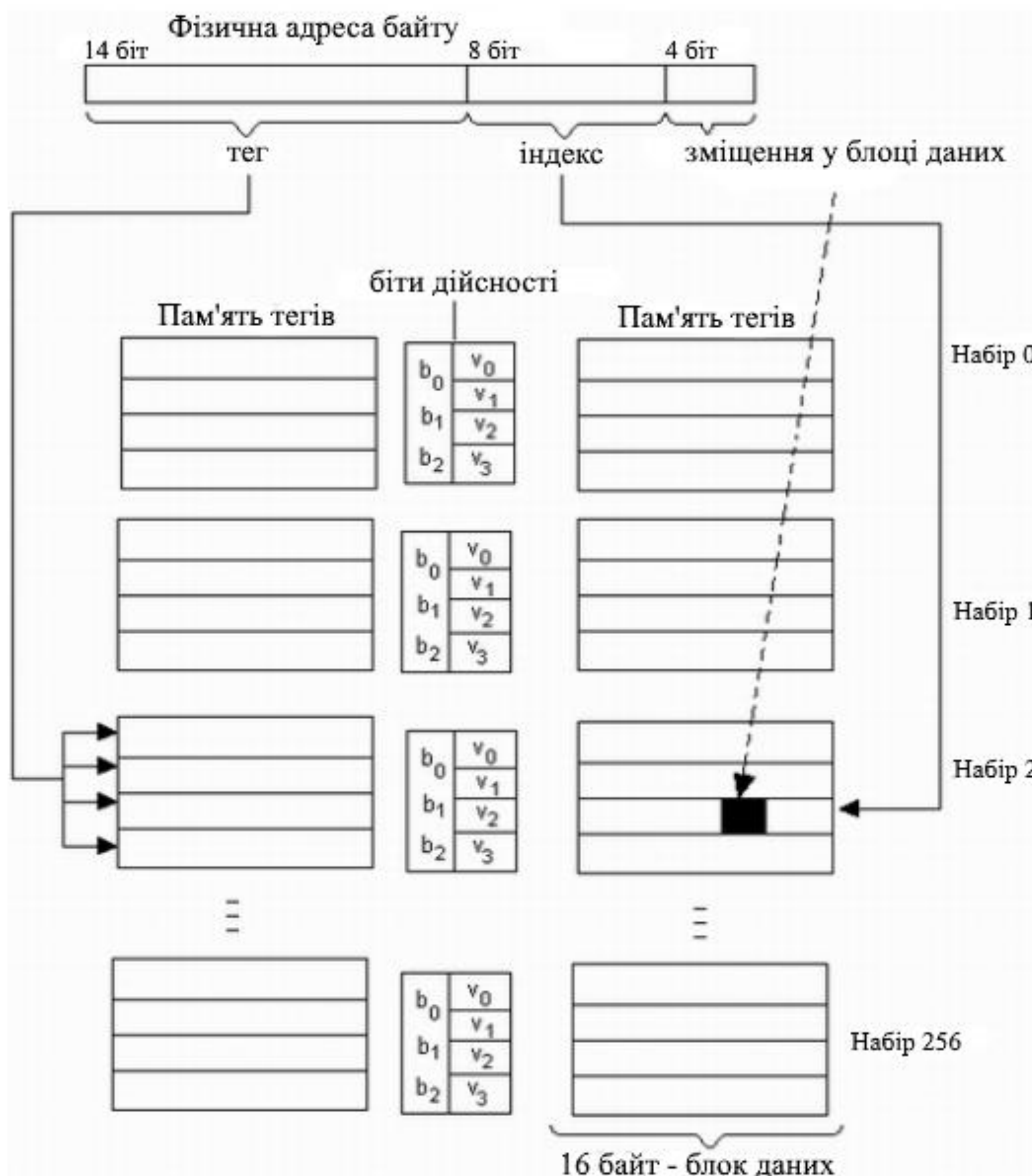


Рисунок 4.18 – Кеш першого рівня процесора Pentium

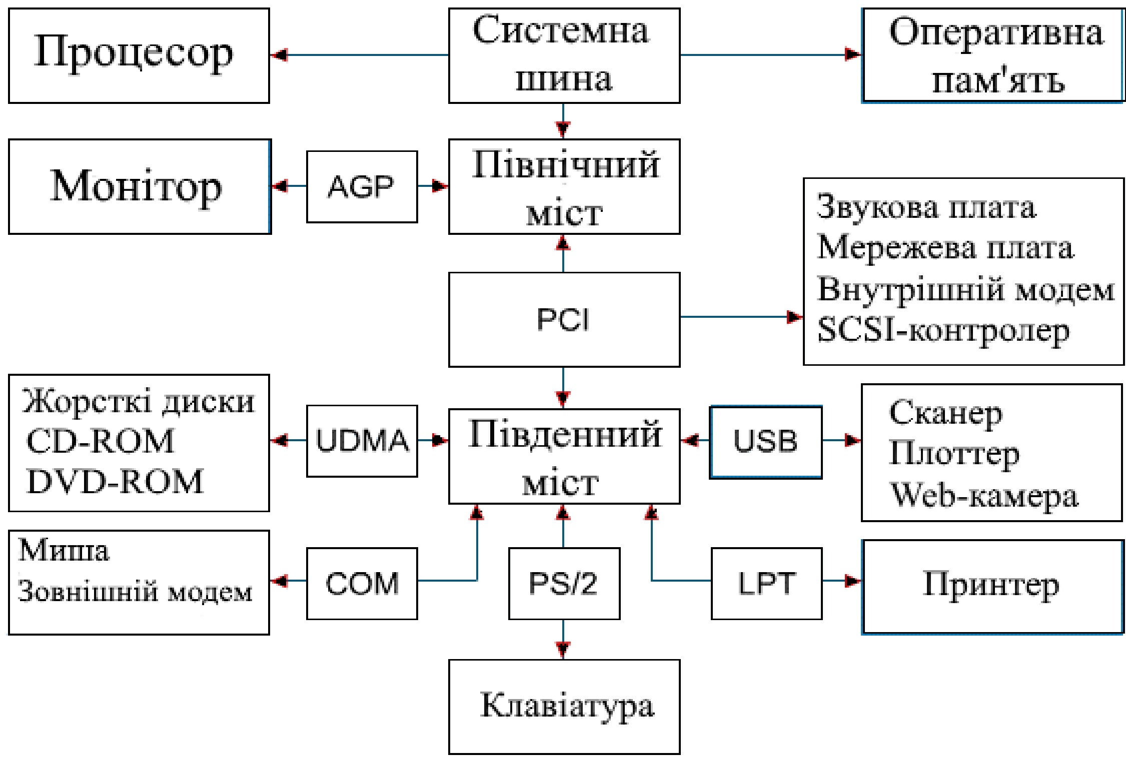
4.8 Шини

Шина – це канал пересилки даних, який використовується спільно різними блоками системи. Шина може бути набором проводникових ліній, витравлених в друкованій платі, дротів, припаяних до роз'ємів, в які вставляються друковані плати, або плоский кабель.

Для узгодження швидкодії на системній платі встановлюються спеціальні мікросхеми (чіпсети), що включають в себе контролер оперативної пам'яті (так званий північний міст) і контролер периферійних пристроїв (рис. 4.19).

Північний міст забезпечує обмін інформацією між процесором і оперативною пам'яттю по системній (головній) шині – ***магістраль*** (рис. 4.20).

До північного мосту підключається шина PCI (Peripherial Component Interconnect bus – шина взаємодії периферійних пристроїв), яка забезпечує обмін інформацією з контролерами периферійних пристроїв.



Риунок. 4.19 – Логічна схема системної плати

Інформація передається шиною у вигляді груп бітів. До складу шини для кожного біта слова може бути передбачена окрема лінія (***паралельна шина***), або усі біти слова можуть послідовно в часі використати одну лінію (***послідовна шина***).

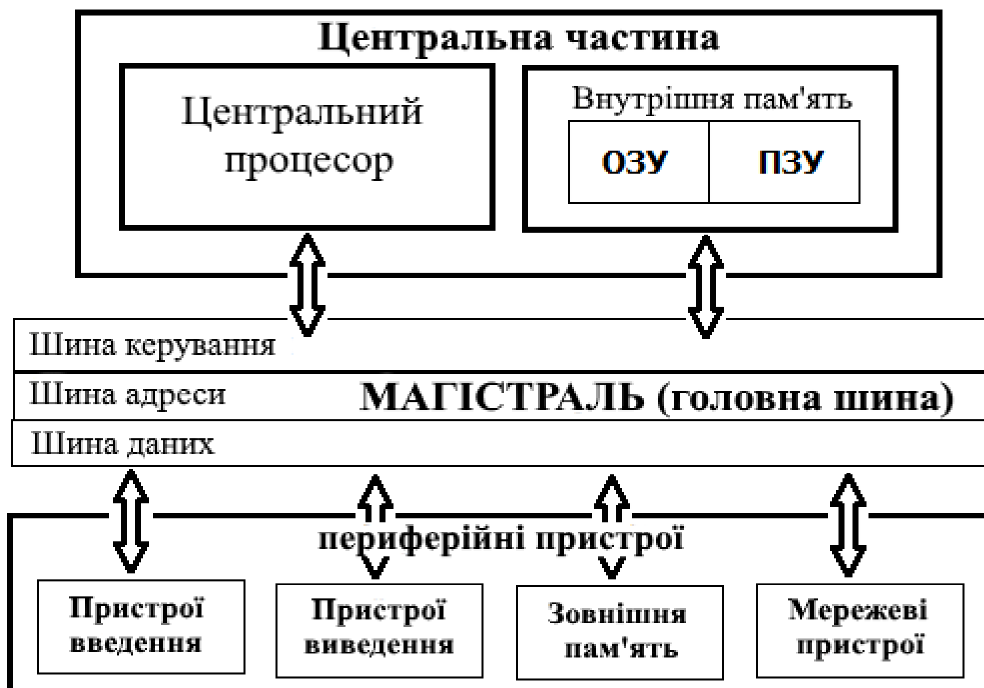


Рисунок 4.20 – Головна шина комп'ютера

Дані на шині призначаються тільки для одного з пристроїв-одержувачів. Поєднання управляючих і адресних сигналів визначає для кого саме призначені дані. Управляюча логіка збуджує спеціальні ***стробуючі сигнали*** (від грецького strobos – кружляння, вихор), щоб вказати одержувачеві, коли йому слід приймати дані. Одержувачі і відправники можуть бути однонапрямленими (передача або прийом) і двонаправленими (передача і прийом).

Внаслідок еволюції сучасна система Pentium містить вісім шин (шина кешу, локальна шина, шина пам'яті, PCI, SCSI, USB, IDE, ISA), кожна зі своєю швидкістю передачі і своїми функціями.

Системи Pentium мають кеш 1-го рівня (L1), вбудований в процесор, і набагато більший кеш 2-го рівня (L2), підключений до процесора окремою шиною. Дві основні шини – це **IDE** (Industry Standard Architecture – промислова стандартна архітектура) і її наступник, шина **PCI** (Peripheral Component Interconnect – інтерфейс периферійних пристроїв). Крім того, в систему входять три спеціалізовані шини: IDE, USB і SCSI. Шина IDE служить для приєднання периферійних пристроїв – дисків, компакт дисків.

Шина **USB** (Universal Serial Bus – універсальна послідовна шина) для приєднання до комп'ютера усіх повільних пристроїв введення-виведення (клавіатура, миша, принтер). Усі USB-пристрої використовують один драйвер, позбавляючи тим самим від необхідності установки нових драйверів для кожного нового USB-пристрою.

SCSI (Small Computer System Interface – системний інтерфейс малих комп'ютерів) – це високопродуктивна шина, вживана для швидких дисків, сканерів і інших пристроїв.

4.9 Технології введення-виведення

Основними компонентами підсистеми введення-виведення є драйвери, керуючими зовнішніми пристроями, і файлова система. В роботі підсистеми введення-виведення активно бере участь диспетчер переривань. Більш того, основне навантаження диспетчера переривань обумовлено саме підсистемою введення-виведення, тому диспетчер переривань іноді вважають частиною підсистеми введення-виведення.

Можливі три методи виконання операцій введення-виведення:

1. Програмоване введення-виведення.
2. Уведення-виведення з використанням переривань.
3. Прямий доступ до пам'яті.

Програмоване уведення-виведення. Коли процесор зустрічає команду, пов'язану з введенням-виведенням, він виконує її, передаючи відповідні команди контролеру введення-виведення. Контролер введення-виведення більше не посилає процесору ніяких сигналів, у тому числі і сигналів переривання. Відповідальність за періодичну перевірку стану модуля введення-виведення несе процесор. Він робить перевірку до тих пір, поки операція введення-виведення не завершиться. Процесор також відповідає за витягання і розміщення даних в пам'яті.

Уведення-виведення з використанням переривань. Проблема програмованого введення-виведення полягає в тому, що процесор повинен довго чекати, поки контролер введення-виведення читатиме або прийматиме нові дані. Під час очікування процесор повинен постійно робити опитування стану модуля введення-виведення, через що падає продуктивність усієї системи.

При альтернативному підході процесор може передати контролеру команду введення-виведення, а потім перейти до виконання іншої роботи. Коли контролер уведення-виведення знову буде готовий обмінюватися даними з процесором, він просигналізує процесору і зажадає, щоб його обслужили.

Прямий доступ до пам'яті. Хоча введення-виведення, кероване перериваннями більш ефективніше, ніж просте програмоване введення-виведення, воно все ще займає багато процесорного часу для передачі даних між пам'яттю і контролером уведення-виведення. При цьому через процесор повинні пройти усі дані, що пересилаються.

Для переміщення великих об'ємів даних може використовуватися ефективніший метод – прямий доступ до пам'яті (Direct Memory Access – **DMA**). Функції DMA виконуються окремим контролером системної шини або можуть бути вбудовані в контролер введення-виведення. Коли процесору треба прочитати або записати блок даних, він генерує команду для модуля DMA, посилаючи йому таку інформацію:

- вказівка на читання або запис;
- адреса пристрою введення-виведення;
- початкова адреса блоку пам'яті;
- кількість слів, які мають бути прочитані або записані.

Передавши повноваження з виконання цих операцій контролеру DMA, процесор продовжує роботу. Контролер DMA передає увесь блок даних в пам'ять або з неї, не задіючи при цьому процесор. Після закінчення передачі контролер DMA посилає процесору сигнал переривання.

Контрольні питання і тести до розділу 4

Контрольні питання

1. На чому засновані засоби підтримки привілейованого (захищеного) режиму?
2. Які операції виконують засоби трансляції адрес і перемикання контексту?
3. Які типи переривань ви знаєте, і яка їх роль в роботі будь-якої операційної системи?
4. Опишіть принцип реалізації і роботи системного таймера.
5. Назовіть машинно-залежні компоненти ОС.
6. Які основні драйвери містить базова система введення-виведення – BIOS?
7. З яких основних структурних компонентів складається комп'ютер?
8. Які регістри комп'ютера доступні користувачеві?
9. Дайте визначення кеш-пам'яті і методу кешування.
10. Опишіть принцип дії кеш-пам'яті.
11. Що таке кеш-попадання і кеш-промах?

12. Що таке просторова і часова локальність?
13. На основі яких двох підходів будується кеш-пам'ять у багатьох сучасних процесорах?
14. Опишіть принцип роботи асоціативного пошуку для кешів з випадковим відображенням даних.
15. У чому полягають відмінності виключень від переривань?

Тести

1. Розташуйте пристрої пам'яті в порядку убування їх швидкості:
   1) жорсткий диск, оперативна пам'ять, кеш-пам'ять, оптичний диск, регістр;
   2) кеш-пам'ять, оперативна пам'ять, жорсткий диск, оптичний диск, регістр;
   3) оперативна пам'ять, кеш-пам'ять, жорсткий диск, оптичний диск, регістр;
   4) регістр, кеш-пам'ять, оперативна пам'ять, жорсткий диск, оптичний диск.
2. Чи є BIOS частиною операційної системи?
   1) так;
   2) ні.
3. Чи впливає розрядність шини на продуктивність системи?
   1) так;
   2) ні.
4. Якщо кешування застосовується для зменшення середнього часу доступу до оперативної пам'яті, то в якості кешу використовують:
   1) буфери в оперативній пам'яті, в яких осідають найактивніше використовувані дані;
   2) буфери в зовнішній пам'яті, в яких осідають найактивніше використовувані дані.
   3) швидкодіючу статичну пам'ять.
5. Якщо кешування використовується системою введення-виведення для прискорення доступу до даних, що зберігаються на диску, то в якості кеша використовують:
   1) буфери в оперативній пам'яті, в яких осідають найактивніше використовувані дані;
   2) буфери в зовнішній пам'яті, в яких осідають найактивніше використовувані дані.
   3) швидкодіючу статичну пам'ять.

5 УПРАВЛІННЯ ПРОЦЕСАМИ

Найважливішою функцією ОС є організація раціонального використання усіх її апаратних та інформаційних ресурсів. До основних ресурсів можуть бути віднесені процесори, пам'ять, зовнішні пристрої, дані і програми. Маючи в розпорядженні одні і ті ж апаратні ресурси, але керована різними ОС, обчислювальна система може працювати з різним ступенем ефективності. Тому знання внутрішніх механізмів операційної системи дозволяє побічно судити про її експлуатаційні можливості і характеристики. Хоча і в однопрограмній ОС необхідно розв'язувати задачі управління ресурсами, головні складнощі на цьому шляху виникають в мультипрограмних ОС, в яких за ресурси конкурують відразу декілька додатків. Саме тому велика частина всіх проблем, що розглядаються тут, стосується також і мультипрограмних систем.

Одним з основних понять, пов'язаних з ОС, є процес – абстрактне поняття, що описує роботу програми [10]. Усе програмне забезпечення, що функціонує на комп'ютері, включаючи і ОС, можна представити набором процесів.

5.1 Поняття процесу

Щоб підтримувати мультипрограмування, ОС повинна визначити і оформити для себе ті внутрішні одиниці роботи, між якими розподілятиметься процесор та інші ресурси комп'ютера. Пояснюючи поняття «Операційна система» і описуючи способи побудови ОС, ми часто застосовували слова «програма» і «завдання». Нині в більшості ОС визначені два типи одиниць роботи: більша одиниця – ***процес***, або ***задача***, і менш велика – ***потік***, або ***нитка***. Причому процес виконується у формі одного або декількох потоків.

При використанні цих термінів часто виникають складнощі. Це відбувається в силу декількох причин. По-перше, – специфіка різних ОС, коли співпадаючі по суті поняття одержали різні назви, наприклад задача (task) в OS/2, OS/360 і процес (process) в UNIX, Windows NT, NetWare. По-друге, у міру розвитку системного програмування і методів організації обчислень деякі з цих термінів отримали нове смислове значення, особливо це стосується поняття «процес», який поступився багатьма своїми властивостями новому поняттю «***потік».*** По-третє, термінологічні складнощі породжуються наявністю декількох варіантів перекладу англомовних термінів. Наприклад, термін ***«thread»*** перекладається як ***«нитка», «потік», «полегшений процес», «мінізадача»*** тощо. Далі в якості назви одиниць роботи ОС використовуватимуться терміни ***«процес»*** і ***«потік».*** У тих же випадках, коли відмінності між цими поняттями не гратимуть істотної ролі, вони об'єднуються під узагальненим терміном «задача».

Раніше говорили: обчислювальна система виконує одну або декілька програм, операційна система планує завдання, програми можуть обмінюватися даними тощо. Використовували ці терміни в деякому загальновживаному, життєвому сенсі, припускаючи, що усі читачі однаково уявляють собі, що мається на увазі під ними в кожному конкретному випадку. При цьому одні і ті ж слова означали і об'єкти в статичному стані, що не обробляються

обчислювальною системою (наприклад, сукупність файлів на диску), і об'єкти в динамічному стані, що знаходяться в процесі виконання. Це було можливо, поки ми говорили про загальні властивості операційних систем, не вдаючись до подробиць їх внутрішнього устрою і поведінки, або про роботу обчислювальних систем 1-2-го поколінь, які не могли обробляти більше однієї програми або одного завдання одночасно, по суті справи не маючи операційних систем. Але тепер ми знайомимося з деталями функціонування сучасних комп'ютерних систем, і нам доведеться уточнити термінологію.

Розглянемо такий приклад. Два студенти запускають програму витягання квадратного кореня. Один хоче обчислити квадратний корінь з 4, а другий – з 9. З точки зору студентів, запущена одна і та ж програма; з точки зору комп'ютерної системи, їй доводиться займатися двома різними обчислювальними процесами, оскільки різні початкові дані призводять до різного набору обчислень. Отже, на рівні того, що відбувається всередині обчислювальної системи ми не можемо використати термін «програма» в призначеному для користувача значенні слова.

Розглядаючи системи пакетної обробки, ми ввели поняття «завдання» як сукупність програми, набору команд мови управління завданнями, необхідних для її виконання, і вхідних даних. З точки зору студентів, вони, підставивши різні початкові дані, сформували два різні завдання. Можливо, термін «***завдання»*** підійде нам для опису внутрішнього функціонування комп'ютерних систем? Щоб з'ясувати це, давайте розглянемо інший приклад. Нехай знову обидва студенти намагаються витягти корінь квадратний з 9, тобто нехай вони сформували ідентичні завдання, але завантажили їх в обчислювальну систему із зрушенням за часом. Тоді як одне з виконуваних завдань приступило до друку отриманого значення, і чекає закінчення операції введення-виведення, друге тільки починає виконуватися. Чи можна говорити про ідентичність завдань усередині обчислювальної системи в даний момент? Ні, оскільки ***стан*** процесу їх виконання різний. Отже, і слово «завдання» в призначеному для користувача сенсі не може застосовуватися для опису того, що відбувається в обчислювальній системі.

Це відбувається тому, що терміни «***програма»*** і ***«завдання»*** призначені для опису статичних, неактивних об'єктів. Програма ж у процесі виконання є динамічним, активним об'єктом. По ходу її роботи комп'ютер обробляє різні команди і перетворює значення змінних. Для виконання програми ОС повинна виділити певну кількість оперативної пам'яті, закріпити за нею певні пристрої введення-виведення або файли, тобто зарезервувати певні ресурси із загального числа ресурсів усієї обчислювальної системи. І, звичайно ж, неможливе виконання програми без надання їй процесорного часу, тобто часу, впродовж якого процесор виконує коди цієї програми.

Кількість ресурсів і конфігурація системи з часом можуть змінюватися. Тому для опису таких активних об'єктів усередині комп'ютерної системи замість термінів «програма» і «завдання» використовується термін – «***процес».*** Таким чином, процес – це контейнер, в якому міститься вся інформація, необхідна для роботи програми [9].Термін «процес» уперше почали застосовувати розробники системи MULTICS в 1960-х роках.

Процес знаходиться під управлінням ОС, тому в ньому може виконуватися не лише код програми, але і частина коду ядра ОС (що не знаходиться у виконуваному файлі), як у випадках, спеціально запланованих авторами програми (наприклад, при використанні системних викликів), так і в непередбачених ситуаціях (наприклад, при обробці зовнішніх переривань). В операційних системах, де існують і процеси, і потоки, процес розглядається ОС як заявка на споживання усіх видів ресурсів, окрім одного – процесорного часу. Цей останній найважливіший ресурс розподіляється ОС між іншими одиницями роботи – ***потоками***, які і одержали свою назву завдяки тому, що вони є послідовностю (потоки виконання) команд.

У простому випадку процес складається з одного потоку, і саме таким чином трактували поняття «процес» до середини 1980-х років (наприклад, в ранніх версіях UNIX) і в такому ж виді він зберігся в деяких сучасних ОС. У таких системах поняття «потік» повністю поглинається поняттям «процес», тобто залишається тільки одна одиниця роботи і споживання ресурсів – процес. Мультипрограмування здійснюється в таких ОС на рівні процесів. Відмітимо, що в однопрограмних системах не виникає необхідності введення поняття, що означає одиницю роботи, оскільки там не існує проблеми розподілу ресурсів. Взаємозв'язок між завданнями, процесами і потоками показаний на рис. 5.1.

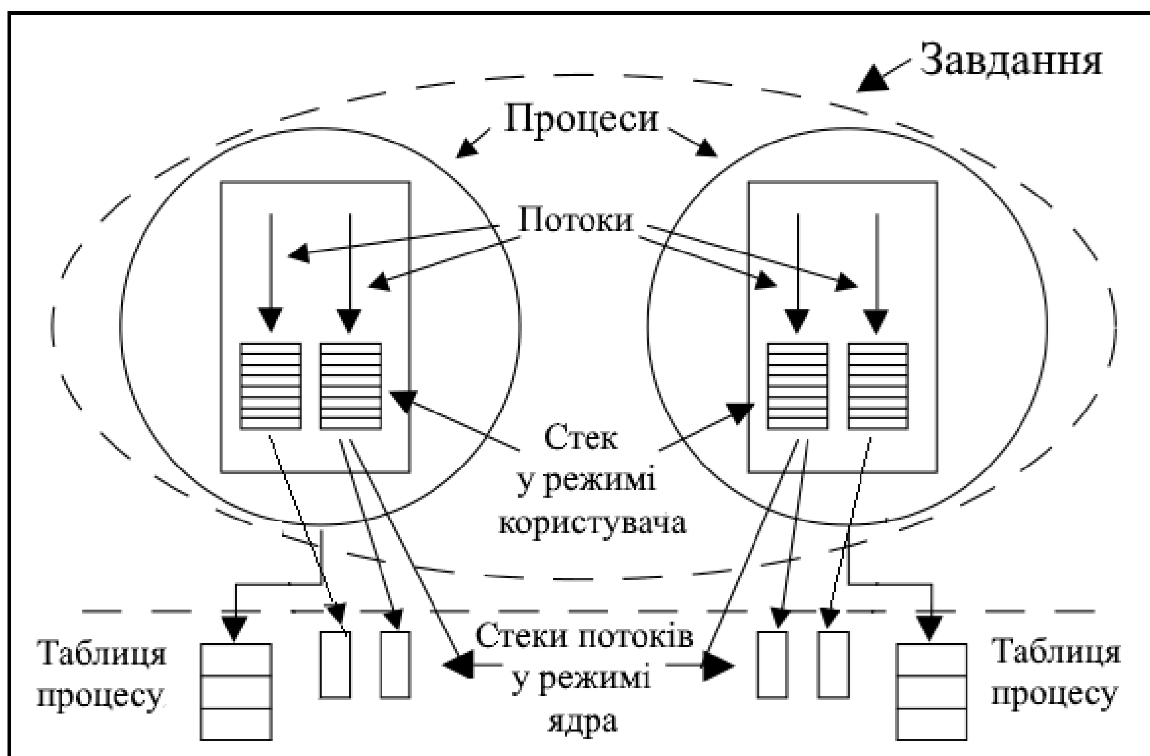


Рисунок 5.1 – Завдання, процеси, потоки

Таким чином, під процесом розуміють програму разом з даними і необхідними ресурсами, яка виконується обчислювальною системою, включаючи поточні значення лічильника команд і регістрів. Тобто, усе програмне забезпечення, у тому числі ОС, які виконуються на комп'ютері, є рядом процесів.

5.2 Поняття ресурсу

Визначення концепції процесу переслідує мету виробити механізм розподілу ресурсів і управління ними. Поняття ресурсу, як і поняття процесу, є, мабуть, основними при розгляді ОС. Ресурси обчислювальної машини діляться на апаратні і програмні (рис. 5.2). Апаратні ресурси – це процесори, запам'ятовуючі пристрої, канали уведення-виведення, периферійні пристрої тощо. Програмними ресурсами є програми і дані, які можуть бути надані користувачам (наприклад, транслятори, бібліотеки стандартних програм, файли, повідомлення тощо).

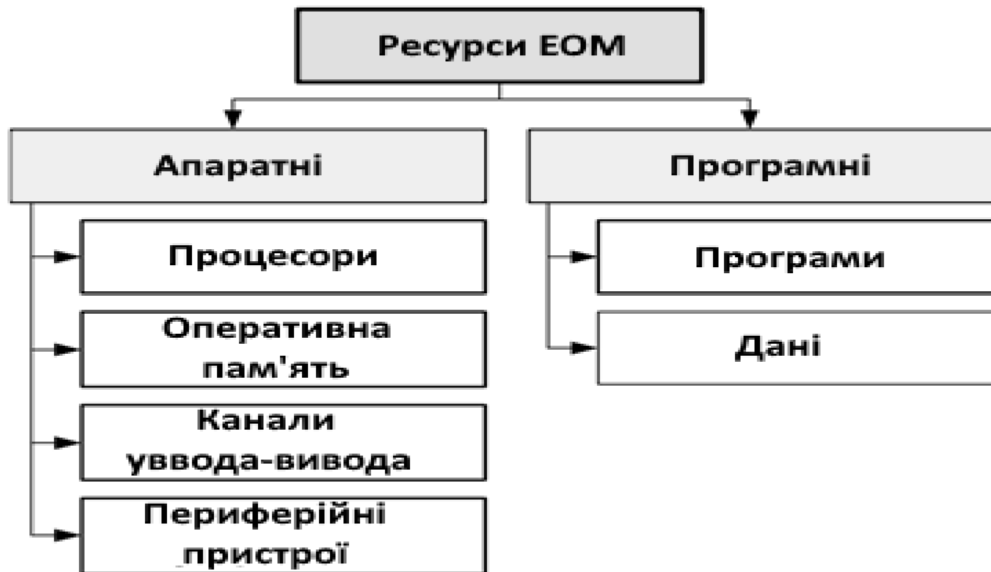


Рисунок 5.2 – Класифікація ресурсів ОЕМ

Термін ресурс застосовується стосовно повторно використовуваних об'єктів, які запитуються, використовуються і звільняються процесами в період їх активності. Іншими словами, ресурсом називається всякий об'єкт, який може розподілятися усередині системи.

Ресурси можуть бути ***спільними***, тобто такими, що ***розділяються***, коли декілька процесів можуть їх використати ***одночасно*** (***паралельно***) в один і той же момент часу або впродовж деякого інтервалу часу процеси використовують ресурс ***почергово.*** Також ресурси можуть бути і ***неподілимими (монопольними)*** (рис. 5.3).

Рисунок 5.3 – Способи використання апаратних ресурсів

При розробці перших ОС ресурсами вважалися процесорний час, пам'ять, канали введення-виведення і периферійні пристрої. Проте дуже скоро поняття ресурсу стало універсальнішим і загальнішим. Різного роду програмні і інформаційні ресурси також можуть бути визначені для системи як об'єкти, які можуть розділятися і розподілятися. Більше того, окрім системних ресурсів, про які ми зараз говорили, як ресурс стали тлумачити і такі об'єкти, як ***повідомлення***, якими обмінюються задачі.

При **монопольному використанні** ресурси закріплюються за одним процесом на весь час його виконання і стають недоступними для інших процесів незалежно від міри їх використання. Такі ресурси називаються ***неподілимими***. До таких неподілимих ресурсів, що допускають тільки монопольне використання, можна віднести, наприклад, друкуючий пристрій, стримери, CD-і DVD-приводи тощо.

Спільне використання ресурсів означає, що декілька процесів розділяють ресурс, якщо цей ресурс впродовж деякого часу знаходиться в розпорядженні одного процесу, потім він може бути відібраний у цього процесу і переданий іншому процесу. Інакше кажучи, ресурс, що розділяється, використовується декількома процесами. Спільне використання такого ресурсу організовується по одному з таких принципів:

- за принципом почергового використання;
- за принципом паралельного використання.

При почерговому використанні ресурс, що розділяється, спільно використовується декількома процесами, почергово переходячи від процесу до процесу. Наприклад, в системі з розподіленням часу п'ять процесів можуть спільно використати один процесор, отримуючи кожний по одній секунді процесорного часу. Практично таким чином відбувається і розподілення інших апаратних ресурсів, різні лише інтервали часу, які надаються процесам.

Спосіб спільного використання ресурсів вимагає наявності спеціальних засобів подолання конфліктних ситуацій і управління чергами запитів, що поступають до спільно використовуваного ресурсу.

При паралельному використанні ресурсом, що розділяється, декілька процесів можуть користуватися одночасно. Прикладом загальнодоступного ресурсу, яким можуть користуватися відразу декілька процесів, може служити оперативна пам'ять.

Монопольно використовувані ресурси називаються ***критичними***, оскільки використання їх поперемінно або паралельно декількома процесами повинно бути виключено, оскільки це може привести до неправильних результатів або до небажаної форми представлення результатів, наприклад, як при поперемінному використанні друкуючого пристрою.

Спосіб **віртуального використання** ресурсів полягає в програмній імітації такого середовища для виконання процесів, в якому обмежені в кількісному відношенні реальні апаратні ресурси як би розширюються. Ресурс, імітований програмними засобами, називають віртуальним. Так, в системі з розподіленням часу з одним процесором імітується середовище, в якому кожен

із процесів має віртуальний процесор, який «працює» дещо повільніше за реальний фізичний процесор.

Транслятори, бібліотеки стандартних програм, прикладні програми – це програмні ресурси, які подібно до апаратних ресурсів, підлягають розподілу. Ці програми зберігаються в зовнішній пам'яті і завантажуються в оперативну пам'ять тільки тоді, коли в них виникає необхідність.

У свою чергу, деякі ресурси, що розділяються, можуть бути ***вивантажуваними ресурсами.*** Зразком такого ресурсу є пам'ять. Розглянемо, наприклад, систему з призначеною для користувача пам'яттю 64 Мб, одним принтером і двома процесами по 64 Мб, кожен з яких хоче щось надрукувати. Завантажується в пам'ять процес *A,* просить і отримує принтер, потім починає обчислювати дані для друку. Ще не закінчивши розрахунки, він перевищує свій квант часу і вивантажується на диск в область підкачки разом з програмою і розрахунковими даними, звільняючи ресурс (пам'ять) для наступного процесу.

Тепер працює процес *B* і безуспішно намагається звернутися до принтера (неподільного ресурсу) і не може продовжити свою роботу без принтера, який зайнятий процесом *A*. У свою чергу, процес *B* займає пам'ять, і жоден з них не може продовжувати роботу без ресурсу, що утримується іншим. На щастя, можна вивантажити (забрати) пам'ять у процесу *B,* перемістивши його на диск в область підкачування і викачавши з диска в пам'ять процес *A.* Тепер процес *A* може закінчити обчислення, виконати друк і звільнити принтер.

Неподільний ресурс, на противагу подільним, – це такий ресурс, який не можна забрати від поточного процессу, не знищивши результати його обчислень (***невивантажений ресурс***). Наприклад, якщо в момент запису на компакт-диск несподівано забрати в процесу пристрій, то в результаті ми отримаємо зіпсований компакт-диск. Послідовність подій, необхідних для використання ресурсу представлена нижче в абстрактній формі:

1) запит ресурсу;
2) використання ресурсу;
3) повернення ресурсу.

Якщо ресурс недоступний, коли він потрібен, то процес, що просить його, змушений чекати. У деяких ОС при невдалому зверненні до ресурсу процес автоматично блокується, і поновлюється тільки після того, як ресурс стає доступним. В інших системах запит ресурсу, що отримав відмову, повертає код помилки, тоді процес може почекати трохи і спробувати ще раз наново зробити запит ресурсу.

Задачей ОС є управління процесами і ресурсами комп'ютера або, точніше, організація раціонального використання ресурсів в інтересах найефективнішого виконання процесів. Для розв'язання цієї задачі операційна система повинна мати в розпорядженні інформацію про поточний стан кожного процесу і ресурсу. Універсальний підхід до надання такої інформації полягає в створенні і підтримці таблиць з інформацією по кожному об'єкту управління (рис. 5.4).

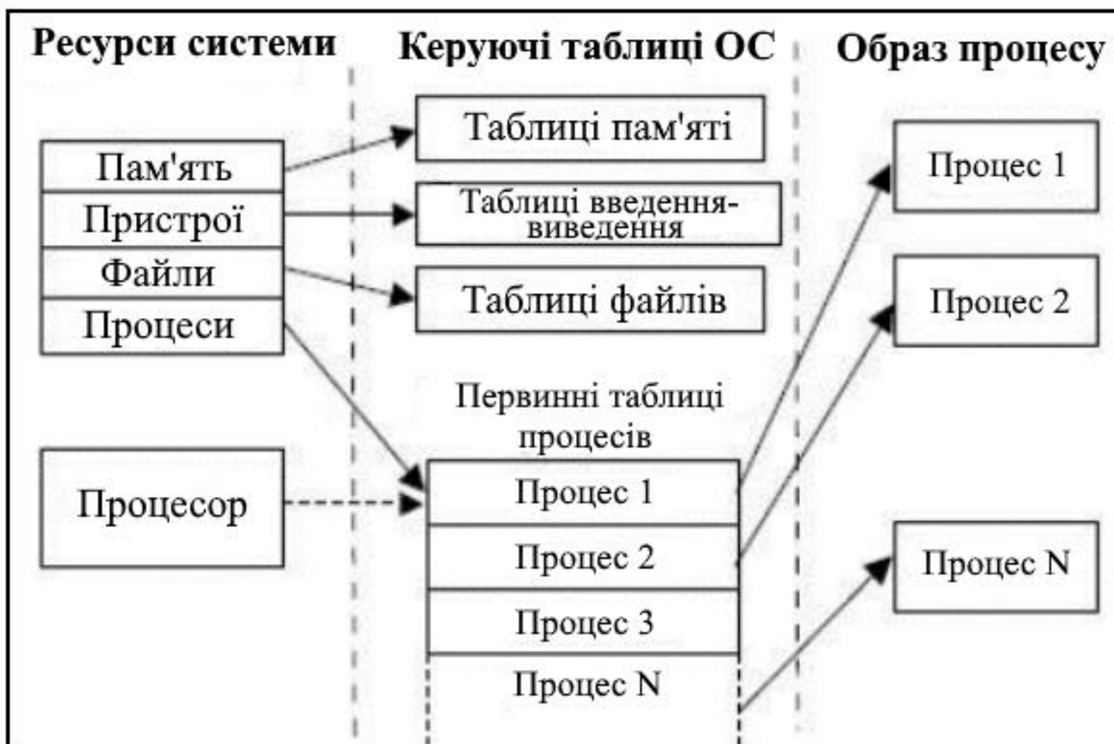


Рисунок 5.4 – Таблиці ОС

5.3 Модель процесу

У цій моделі все програмне забезпечення, що функціонує на комп'ютері, іноді включаючи власне операційну систему, організоване у вигляді набору послідовних процесів. З таких позицій абстрактної моделі кожен процес має свій віртуальний процесор. Насправді реальний процесор перемикається з процесу на процес, представляючи кожному процесу від десятків до сотень мілісекунд, створюючи ілюзію їх паралельного виконання (***псевдопаралельність*).** Таке перемикання називається ***багатозадачністю*** або ***мультипрограмуванням***. У багатопроцесорній системі декілька процесів можуть не лише чергуватися, але і виконуватися одночасно.

У перших обчислювальних системах будь-яка програма могла виконуватися тільки після повного завершення попередньої, оскільки ці системи були побудовані відповідно до принципів архітектури Яноша Джона фон Неймана. У таких комп'ютерах усі підсистеми і пристрої управлялися виключно центральним процесором. Центральний процесор здійснював і виконання обчислень, і управління операціями введення-виведення даних. Природно, поки здійснювався обмін даними між оперативною пам'яттю і зовнішніми пристроями, процесор не міг виконувати обчислення. Введення до складу обчислювальної машини спеціальних контролерів дозволило поєднати в часі (розпаралелювати) операції введення-виведення і подальші обчислення на центральному процесорі. Проте процесор все одно продовжував часто і довго простоювати, чекаючи завершення операції введення-виведення. Тому було запропоновано організувати мультипрограмний (мультизадачний) режим.

При мультипрограмуванні підвищується пропускна спроможність системи, але окремий процес ніколи не може бути виконаний швидше, ніж якби він виконувався в однопрограмному режимі – всякий розподіл ресурсів уповільнює роботу одного з учасників за рахунок додаткових витрат на очікування звільнення ресурсу.

Як уже відзначалося, ОС підтримує мультипрограмування (мультипроцесність) і намагається ефективно використати ресурси шляхом організації черг запитів до них. Загальна схема виділення ресурсів така. При необхідності використати який-небудь ресурс (оперативну пам'ять, пристрої введення-виведення, масив даних тощо) задача звертається до супервізора ОС – її центрального управляючого модуля, який може складатися з декількох модулів, наприклад, супервізор введення-виведення, супервізор переривань, диспетчер задач тощо, і повідомляє про свою вимогу. При цьому вказується вид ресурсу і, якщо потрібно, його об'єм (наприклад, кількість елементів пам'яті, кількість доріжок або секторів на диску).

Директива звернення до ОС передає їй управління, перевівши процесор в привілейований режим роботи (режим супервізора). Ресурс може бути виділений задаче, якщо:

– він вільний і в системі немає запитів від задач з вищим пріоритетом;
– поточний запит і раніше видані запити допускають спільне використання ресурсів;
– ресурс використовується задачей нижчого пріоритету і може бути тимчасово відібраний (ресурс, що розділяється).

Отримавши запит, ОС або задовольняє його і повертає управління задаче, або, якщо ресурс зайнятий, ставить задачу в чергу до ресурсу, переводячи її в стан очікування (блокуючи).

Після закінчення роботи з ресурсом задача знову повідомляє ОС про відмову від ресурсу за допомогою спеціального виклику супервізора (за допомогою відповідної директиви), або ОС забирає ресурс сама. Супервізор, отримавши управління, звільняє ресурс і перевіряє, чи є черга до ресурсу, що звільнився. Якщо черга є, то залежно від прийнятої дисципліни обслуговування (правила обслуговування) і пріоритетів заявок він виводить із стану задачу, що чекає ресурс, і переводить його в стан готовності до виконання. Після цього управління передається даній задаче, або тій задаче, яка тільки що звільнила ресурс.

При видачі запиту на ресурс задача може вказати, чи хоче вона володіти ресурсом монопольно або допускає спільне використання. Наприклад, з файлом можна працювати монопольно, а можна і спільно з іншими задачами.

Розбіжність між програмою і процесом полягає в тому, що процес окрім програми, даних і необхідних ресурсів має ще і **стан**. Один процесор може перемикатися між різними процесами, використовуючи деякий алгоритм планування для визначення моменту перемикання з одного процесу до іншого.

5.4 Створення процесу

Створити процес – це, передусім, означає створити описувач процесу, яким виступає одна або декілька інформаційних структур, що містять усі відомості про процес, необхідні операційній системі для управління ним. В число таких відомостей (атрибутів) можуть входити, наприклад, ідентифікатор процесу, дані про розташування в пам'яті виконуваного модуля, міра привілейованості процесу (пріоритет і права доступу) тощо. Прикладами описувачів процесу є [10; 17]:

- блок управління задачою (TCB –Task Control Block) в OS/360;
- блок процесу (PCB – Process Control Block), що управляє, в OS/2;
- дескриптор процесу в UNIX;
- об'єкт-процес (object-process) в Windows NT/2000/2003.

Створення описувача процесу знаменує собою появу в системі ще одного претендента на обчислювальні ресурси. Починаючи з цього моменту при розподілі ресурсів ОС повинна брати до уваги потреби нового процесу.

Створення процесу включає завантаження кодів і даних виконуваної програми цього процесу з диска в оперативну пам'ять. Для цього ОС повинна виявити місце розташування такої програми на диску, перерозподілити оперативну пам'ять і виділити пам'ять виконуваній програмі нового процесу. Потім необхідно виконувати програму у виділеній для неї ділянці пам'яті і, можливо, змінити параметри програми залежно від розміщення її в пам'яті. Крім того, при роботі програми використовується стек, за допомогою якого реалізуються виклики процедур і передача параметрів.

У системах з віртуальною пам'яттю в початковий момент може завантажуватися тільки частина кодів і даних процесу, з тим щоб «підкачувати» інші в міру необхідності. Існують системи, в яких на етапі створення процесу не вимагається неодмінно завантажувати коди і дані в оперативну пам'ять. Замість цього виконуваний модуль копіюється з того каталогу файлової системи, в якому він спочатку знаходився, в область підкачування – спеціальну область диска, відведену для зберігання кодів цих процесів. При виконанні всіх цих дій підсистема управління процесами тісно взаємодіє з підсистемою управління пам'яттю і файловою системою.

Множина, в яку входять програма, дані, стеки і атрибути процесу, називається ***образом процесу***, типовими елементами якого є:

- дані користувача – змінювана частина призначеного для користувача адресного простору (ці програми, стек користувача, модифікований код);
- призначена для користувача програма – програма, яку необхідно виконати;
- системний стек – один або декілька системних стеків для зберігання параметрів і адрес виклику процедур і системних служб;
- управляючий блок процесу – дані, необхідні операційній системі для управління процесом.

Місцезнаходження образу процесу залежить від використовуваної схеми управління пам'яттю. У більшості сучасних ОС з віртуальною пам'яттю образ процесу складається з набору блоків (сегменти, сторінки або їх комбінація), не

обов'язково розташованих послідовно. Така організація пам'яті дозволяє мати в основній пам'яті лише частину образу процесу (активна частина), тоді як у вторинній пам'яті знаходиться повний образ.

ОС повинна мати засіб, який дозволяє переконатися в наявності всіх необхідних процесів. У простіших ОС це можливо реалізувати у вигляді послідовного (пакетного) їх виконання. Універсальні ОС повинні мати засіб створення і переривання процесів в міру необхідності. Розглянемо деякі з можливих засобів вирішення цієї проблеми. Нижче наведені основні події, які призводять до створення процесів.

1. Ініціація системою.
2. Виконання виданого працюючим процесом системного запиту на створення процесу.
3. Запит користувача на створення процесу. У систему з терміналу, наприклад, входить новий користувач.
4. Ініціація пакетної задачі.

При завантаженні ОС створюється декілька процесів. Деякі з них є високо пріоритетними, тобто такими, що забезпечують взаємодію з користувачем і виконують певну функцію. Інші процеси є ***фоновими***, вони не пов'язані з конкретними користувачами, але виконують особливі функції. Так, один фоновий процес може бути призначений для обробки електронної пошти і активізується лише в міру появи листів. Інший фоновий процес може обробляти запити до WEB-сторінок, які розташовані на комп'ютері і активізувалися для обслуговування отриманого запиту. Фонові процеси, які пов'язані з електронною поштою, WEB-сторінками, новинами, виведенням на друк і так далі, мають назву ***демонів***.

Процеси можуть створюватися як під час завантаження ОС, так і пізніше. Новий процес (чи декілька) можуть бути створені на вимогу поточного процесу. Створення нових процесів особливо корисне в тих випадках, коли глобальна задача, яка виконується в комп'ютерній системі, можна сформувати як набір пов'язаних, але незалежних процесів, які взаємодіють між собою.

З технічної точки зору в усіх перерахованих випадках новий процес формується однаково: поточний процес виконує системний запит на створення нового процесу. В ролі поточного процесу може бути процес користувача, системний процес тощо. У будь-якому випадку цей процес виконує тільки системний запит і створює новий процес.

5.5 Завершення процесу

Після того як процес був створений, він починає виконуватися. Але рано чи пізно він завершується через виникнення таких подій:

- звичайний вихід;
- вихід через помилку;
- вихід через фатальну помилку (примусово);
- знищення іншим процесом (примусово);
- перевищення ліміту часу, відведеного програмі;

– недостатній об'єм пам'яті;
– невірна команда;
– втручання оператора.

В основному процеси завершуються в міру їх виконання.

Другою причиною завершення процесу може бути неусувна помилка. Наприклад, якщо користувач набрав на клавіатурі команду *Edit abc.txt,* то редактор *Edit* може закінчити роботу, якщо файлу *abc.txt* не існує.

Третьою причиною завершення процесу може бути помилка, викликана самим процесом, найчастіше пов'язана з помилкою в програмі.

Четвертою причиною завершення процесу може служити виконання іншим процесом системного запиту на знищення процесу.

5.6 Стани процесів

Необхідно відрізняти системні управляючі процеси, які представляють роботу супервізора ОС і займаються розподілом та управлінням ресурсів, від усіх інших процесів: системних оброблювальних процесів, які не входять в ядро ОС, і процесів користувача. Для системних управляючих процесів у більшості ОС ресурси розподіляються спочатку і однозначно. Тому виконання системних управляючих програм не прийнято називати процесами.

У якому стані знаходиться процес залежить від планування обчислювального процесу. Усі види планування, використовувані в сучасних ОС, будуть детальніше розглянуті в розділі «Планування процесів». Види планування залежно від часового масштабу, діляться на ***довгострокове***, ***середньострокове***, ***короткострокове*** і ***планування введення-виведення***.

Розглядаючи частоту роботи планувальника, можна сказати, що довгострокове планування виконується порівняно нечасто, середньострокове дещо частіше. Короткостроковий планувальник, який часто називають ***диспетчером*** (dispatcher), працює, визначаючи, який процес або потік виконуватиметься наступним. Нижче наведений короткий перелік функцій, що виконуються планувальником кожного виду.

1. Довгострокове. Рішення про додавання процесу в пул виконуваних в системі процесів.
2. Середньострокове. Рішення про додавання процесу до числа процесів, повністю або частково розміщених в основній пам'яті.
3. Короткострокове. Рішення про те, який з доступних процесів виконуватиметься процесором.
4. Планування введення-виведення. Рішення про те, який із запитів процесів (потоків) на операцію введення-виведення виконуватиметься вільними пристроями введення-виведення.

Процес може знаходитися в активному і пасивному стані. В ***активному стані*** процес може брати участь у конкуренції за використання ресурсів, а в ***пасивному*** – він тільки відомий системі, але не бере участі в конкуренції (хоча його існування в системі зв'язане з представленням йому оперативної і/або

зовнішньої пам'яті). У свою чергу, активний процес може знаходитися в одному з трьох основних станів (рис. 5.5):

- ***виконання*** – усі ресурси, що зажадалися процесом, виділені. У цьому стані в кожен момент часу може знаходитися тільки один процес, якщо йдеться про однопроцесорну обчислювальну систему;
- ***готовності до виконання*** – ресурси надані, працездатний, але тимчасово процес призупинений, щоб дати можливість виконання іншому процесу;
- ***блокування*** або ***очікування*** – ресурси, що зажадалися, не можуть бути надані, або не завершена операція введення-виведення.

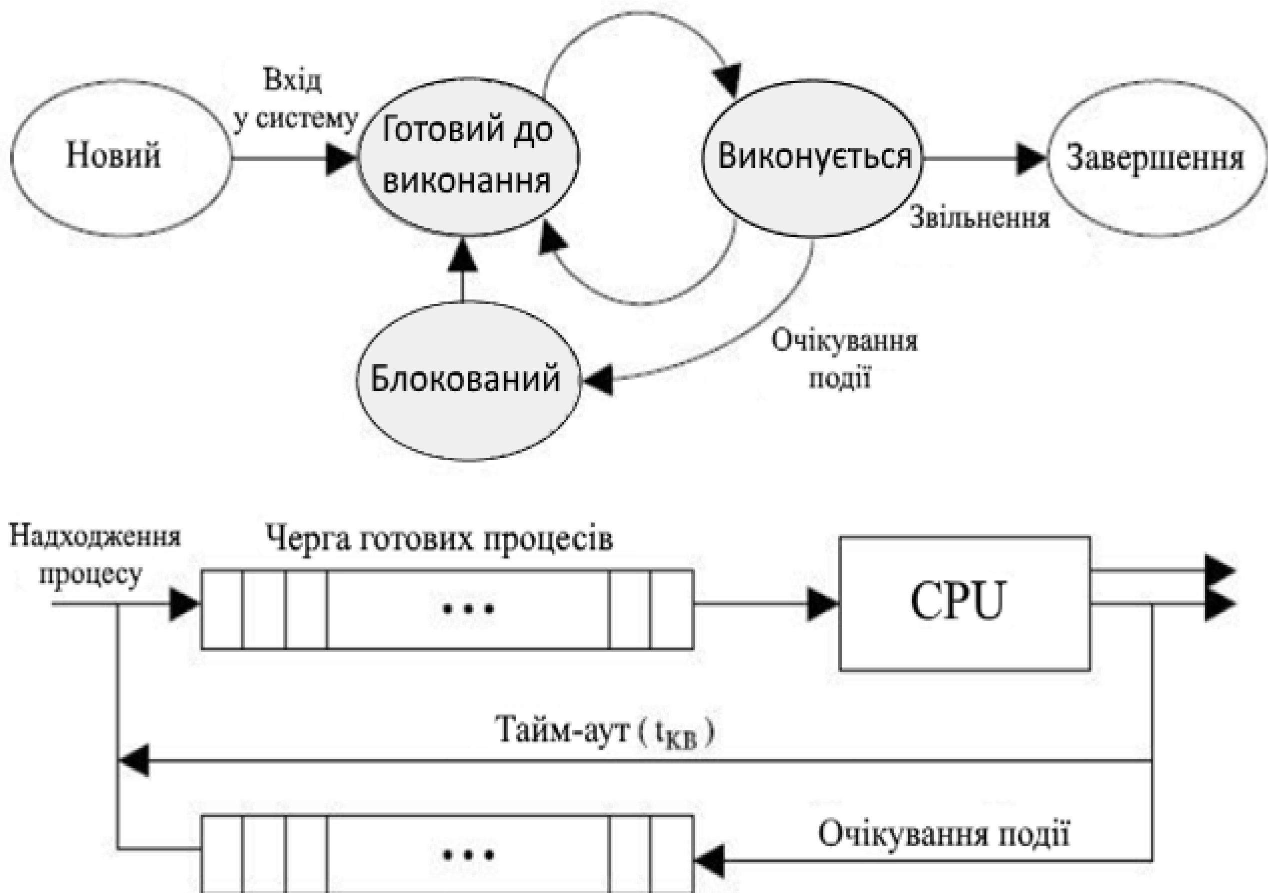


Рисунок 5.5 – Діаграма станів процесу

У стані ***виконання*** в однопроцесорній системі може знаходитися тільки один процес, а в кожному із станів ***очікування*** і ***готовності*** – декілька процесів. Ці процеси утворюють черги відповідно очікуючих і готових процесів.

Як показано на рис. 5.5, між цими трьома станами можуть бути чотири переходи. Життєвий цикл процесу розпочинається зі стану ***готовності***, коли процес готовий до виконання і чекає своєї черги (працездатний, але тимчасово призупинений, щоб дати можливість виконання іншому процесу).

При активізації процес переходить в стан ***виконання*** і знаходиться в ньому до тих пір, поки він сам звільнить процесор, перейшовши в стан ***очікування*** якої-небудь події. В стан очікування він може також перейти за допомогою системного виклику.

Процес може бути насильно «витіснений» з процесора, наприклад, унаслідок вичерпання відведеного цьому процесу кванта процесорного часу. В останньому випадку процес повертається в стан ***готовність.*** У цей же стан процес переходить із стану ***очікування*** після того, як очікувана подія станеться.

У більшості ОС стан ***блокування (очікування)***, у свою чергу, поділяється на певну кількість станів очікування, що відповідають певному виду ресурсу, через відсутність якого процес переходить в заблокований стан.

5.7 Призупинені процеси

Три основні стани процесів, описані раніше (готовий, виконується і блокований), дозволяють змоделювати поведінку процесів і отримати уявлення про реалізацію ОС. Багато ОС створені на основі тільки цих трьох станів.

Можна навести переконливі аргументи на користь додавання в модель процесів і інших станів. Щоб усвідомити, які переваги можуть дати ці нові стани, розглянемо систему, яка не використовує віртуальну пам'ять, в якій кожен процес перед виконанням потрібно завантажити в основну пам'ять. Таким чином, усі процеси, які представлені моделлю з 3-ма станами, повинні знаходитися в основній пам'яті.

Причиною розробки такої системи було повільне, в порівнянні з обчисленнями, виконання операції введення-виведення, яке призводило до простоїв CPU в однозадачній системі. Але організація роботи відповідно до схеми з трьома станами повністю цю проблему не вирішує. Звичайно, при роботі з такою моделлю в пам'яті знаходиться декілька процесів, і доки одні процеси чекають завершення операції введення-виведення, процесор може перейти до виконання інших процесів. Але процесор працює настільки швидше за виконання операцій введення-виведення, що незабаром усі процеси, що знаходяться в пам'яті, виявляються в стані очікування.

Таким чином, CPU може простоювати навіть у багатозадачній системі. Якщо збільшити розмір основної пам'яті для розміщення в ній більшої кількості процесів, то це, по-перше, призведе до подорожчання пам'яті і системи в цілому, а по-друге, сучасні технології створення програм дозволяють створювати все складніші і тому великі за розміром програми.

Іншим рішенням проблеми є ***свопінг,*** який включає перенесення частини процесів з основної пам'яті на диск. Якщо в основній пам'яті немає жодного готового до виконання процесу, ОС переводить один з блокованих процесів на диск (здійснює його свопінг), розміщуючи його в чергу ***призупинених*** (***блокованих***) ***процесів***, які тимчасово витягнуті з основної пам'яті. Далі ОС завантажує інший процес з черги призупинених, після чого продовжує його виконання.

Але свопінг сам по собі є операцією введення-виведення, тому існує ризик погіршити ситуацію, замість її удосконалення. Завдяки тому, що обмін з диском виконується швидше за інші операції введення-виведення (наприклад, виведення на друк), то свопінг найчастіше підвищує продуктивність системи в цілому.

Якщо в модель процесів ввести свопінг, то необхідно ввести новий стан – ***стан призупиненого процесу***. Коли всі процеси в основній пам'яті знаходяться в блокованому стані, ОС може призупинити один з процесів, переводячи його в призупинений стан. Простір, що звільнився в основній пам'яті, потім використовується для завантаження іншого процесу.

Після того, як ОС вивантажила один з процесів на диск, вона має дві можливості вибору процесу для завантаження в основну пам'ять: вона може або створити новий процес, або завантажити процес, який був призупинений перед цим. Може здатися, що краще було б завантажити для обробки раніше призупинений процес, що не призведе до збільшення навантаження на систему.

Але це не зовсім так. Усі процеси, перш ніж були призупиненими, знаходилися в блокованому стані. Зрозуміло, що повернення в пам'ять блокованого процесу не дає ніяких результатів, оскільки він все одно не готовий до виконання. Адже кожен процес в призупиненому стані блокований в очікуванні на якусь певну подію. Якщо ця подія відбувається, процес перестає бути блокованим і можна продовжити його виконання.

Це слід врахувати при розробці ОС. Існують дві незалежні ситуації: чи чекає процес якої-небудь події (тобто, блокований він або ні); чи вивантажений процес з основної пам'яті (тобто, призупинений він чи ні). Маємо 2x2 можливі комбінації, тому необхідні чотири перераховані нижче стани.

1. **Готовий**. Процес, який знаходиться в основній пам'яті і готовий до виконання.
2. **Блокований**. Процес, який знаходиться в основній пам'яті і чекає на певну подію.
3. **Блокований/Призупинений**. Процес, який знаходиться в зовнішній пам'яті (диск) і чекає на певну подію.
4. **Готовий/Призупинений**. Процес, який знаходиться в зовнішній пам'яті (диск), але готовий до виконання, його потрібно тільки завантажити в основну пам'ять.

Перш ніж скласти діаграму переходів станів, де враховуються два нові призупинені стани, потрібно згадати ще одну обставину. Досі ми не враховували наявність віртуальної пам'яті. Вважалося, що процес знаходиться або повністю в основній пам'яті, або повністю в зовнішній пам'яті. За наявності віртуальної пам'яті з'являється можливість виконувати процес, який завантажений в основну пам'ять частково.

Якщо відбувається звернення до відсутнього в основній пам'яті адреси процесу, то ця частина процесу може бути завантажена. Здавалося б, що використання віртуальної пам'яті позбавляє необхідності явного свопінгу, оскільки будь-яку потрібну адресу будь-якого процесу можна перенести в основну пам'ять або з неї за допомогою апаратного забезпечення CPU, керуючого пам'яттю.

Проте, як дізнаємося далі щодо віртуальної пам'яті, за наявності досить великої кількості активних процесів, які повністю або частково знаходяться в основній пам'яті, продуктивність віртуальної пам'яті може виявитися недостатньою. Тому, навіть за наявності віртуальної пам'яті, ОС час від часу

вимагається явно і повністю вивантажувати процеси з основної пам'яті заради підвищення загальної продуктивності.

На рис. 5.6 показана діаграма станів процесу, модифікована з урахуванням операцій призупинення і відновлення.

Рисунок 5.6 – Діаграма стану процесу з операціями призупинення і відновлення

У діаграму введені два нові стани, а саме «призупинений_готовий» і «призупинений_блокований», стану «призупинений_виконується» не буває.

Ініціатором призупинення може бути або сам процес, або інший процес. В однопроцесорній машині процес, що виконується, не може працювати одночасно з ним, щоб видати сигнал призупинення. У мультипроцесорній машині процес, що виконується, може бути призупинений і іншим процесом, що виконується на іншому процесорі.

Процес, що знаходиться в стані готовності, може бути призупинений тільки іншим процесом.

Розглянемо детальніше діаграму станів процесу з операціями призупинення і відновлення, найважливіши з яких наводяться нижче.

Блокований $\rightarrow$ Блокований/Призупинений. Якщо до виконання не готовий жоден процес, то, принаймні, один блокований процес вивантажується з пам'яті, щоб звільнити місце для іншого процесу, який не є блокованим. Цей перехід можна виконувати і за наявності готових до виконання процесів, якщо ОС визначить, що для процесу, що виконується зараз, або процесу, управління до якого перейде найближчим часом, треба збільшити об'єм основної пам'яті для забезпечення високої продуктивності.

Блокований/Призупинений → Готовий/Призупинений Процес у стані блокованого/призупиненого переходить у стан готового до виконання призупиненого процесу, якщо відбувається подія, якої чекав цей процес. Для цього необхідно, щоб ОС мала доступ до інформації про стан призупинених процесів.

Готовий/Призупинений → Готовий. Коли в основній пам'яті немає готових до виконання процесів, ОС для продовження обчислень вимагається завантажити процес в пам'ять. Може статися і так, що в готового до виконання призупиненого процесу більш високий пріоритет, ніж у будь-якого іншого з готових до виконання процесів. У такій ситуації розробник ОС може вирішити, що важливіше забезпечити пріоритет процесу, чим мінімізувати свопінг.

Готовий → Готовий/Призупинений. ОС вважає за краще призупиняти не готовий до виконання процес, а заблокований процес, оскільки до виконання готового процесу можна приступити негайно, а блокований процес тільки даремно займає основну пам'ять, оскільки не може бути виконаний. Але іноді виявляється, що єдиний спосіб звільнити досить великий розмір основної пам'яті – це призупинити готовий до виконання процес. ОС може також замість блокованого процесу з вищим пріоритетом призупинити готовий до виконання процес з нижчим пріоритетом, якщо блокований процес досить скоро буде готовий до виконання.

Новий → Готовий/Призупинений і Новий → Готовий (на рисунку не зображений). Після створення нового процесу цей процес може бути доданий або в чергу готових до виконання процесів, або в чергу готових до виконання призупинених процесів. У будь-якому з цих випадків ОС повинна створити таблиці для управління процесом і виділити йому адресний простір. Краще виконувати ці дії на ранніх етапах, щоб мати більший запас неблокованих процесів. Але якщо дотримуватися цієї стратегії, то в основній пам'яті може не вистачити місця для нового процесу. З цієї причини передбачений перехід нового процесу в стан призупиненого готового до виконання. З іншого боку, створення процесу в «останню мить» призводить до зменшення непродуктивних витрат і дозволяє ОС виконувати свої обов'язки зі створення процесів навіть тоді, коли вона переповнена блокованими процесами.

Блокований/Призупинений → Блокований. На перший погляд може здатися, що враховувати такий перехід немає сенсу. Для чого завантажувати в пам'ять процес, який не готовий до виконання? Але розглянемо ситуацію: завершився деякий процес, звільнивши при цьому певну частину основної пам'яті. У черзі заблокованих призупинених процесів знаходиться процес, пріоритет якого вищий, ніж у будь-якого процесу з черги готових до виконання, але призупинених процесів. Крім того, ОС має в розпорядженні аргументи на користь того, що скоро станеться подія, яка зніме блокування з цього високопріоритетного процесу. При таких обставинах доречно віддати перевагу блокованому процесу перед готовим до виконання, завантаживши в основну пам'ять саме його.

Виконується → Готовий/Призупинений. Процес, що виконується, в якого вийшов відведений йому час, переходить в стан готового до виконання. Проте за наявності процесу з вищим пріоритетом, який знаходиться в черзі блокованих призупинених процесів, і тільки що був розблокований, ОС може віддати перевагу саме йому. Щоб звільнити частину основної пам'яті, вона може перевести процес, що виконується, безпосередньо в стан готового до виконання призупиненого процесу.

Довільний стан → Завершення (на рисунку не зображений). Завершується процес, що виконується зараз, – це відбувається або через те, що він виконаний до кінця, або через помилки при виконанні. Але в деяких ОС процес може завершитися процесом, що створив його, або разом із завершенням батьківського процесу. Таке завершення можливе за умови, що процеси з будь-якого стану можуть переходити в стан завершення.

У конкретних операційних системах стани процесу можуть бути ще більше деталізовані, можуть з'явитися деякі нові варіанти переходів з одного стану в інший. Так, наприклад, модель станів процесів для операційної системи Windows NT містить 7 різних станів, а для операційної системи Unix – 9. Проте так чи інакше, усі операційні системи підпорядковуються викладеній вище моделі.

5.8 Опис процесів

Операційна система управляє подіями, які відбуваються в комп'ютерній системі. Вона планує і координує виконання процесів, виділяє їм ресурси і надає за запитом системних і призначених для користувача програм основні сервіси. Можна представити ОС як деякий механізм, який керує тим, як процеси використовують системні ресурси.

Ця концепція проілюстрована на рис. 5.7. Нехай в багатозадачному середовищі є декілька процесів (P1, P2,.,Pn), які вже створені і завантажені у віртуальну пам'ять. Кожному процесу для його функціонування потрібний доступ до певних ресурсів. У ситуації, зображеній на рис. 5.7, процес P1 знаходиться в стані виконання, тобто в основній пам'яті знаходиться принаймні частина цього процесу. Крім того, він здійснює управління двома пристроями введення-виведення. Процес P2 теж знаходиться в основній пам'яті, але він блокований, чекаючи, поки звільнитися пристрій введення-виведення, зайнятий процесом P1. Процес Pn вивантажений з основної пам'яті і, відповідно, призупинений.

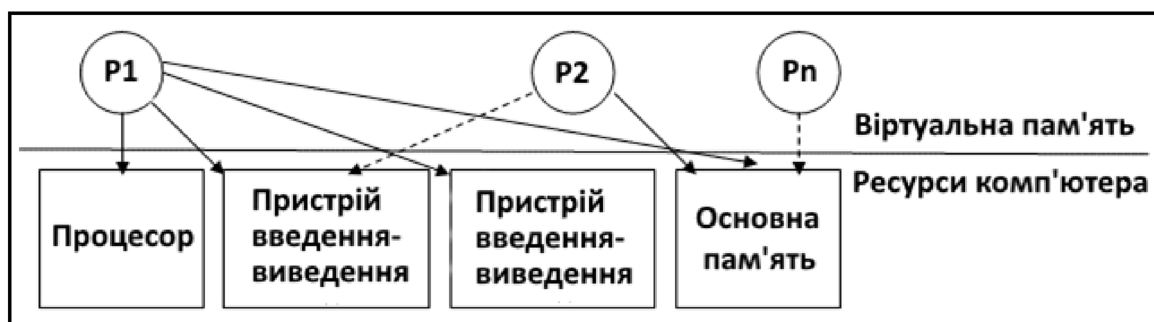


Рисунок 5.7 – Процеси і ресурси в певний момент часу

5.8.1 Управляючі структури ОС

Оскільки в задачі ОС входить управління процесами і ресурсами, вона повинна мати в розпорядженні інформацію про поточний стан кожного процесу і ресурсу. Універсальний підхід до надання такої інформації простий: ОС створює і підтримує таблиці з інформацією по кожному об'єкту управління. Хоча деталі таблиць в різних ОС можуть відрізнятися, по суті, усі ОС підтримують інформацію за чотирма категоріями. P_1 P_2

Таблиці пам'яті (Memory tables) використовуються для того, щоб стежити за основною (реальною) і вторинною (віртуальною) пам'яттю. Процеси, які знаходяться у вторинній пам'яті, використовують деякий різновид віртуальної пам'яті або простий механізм свопінгу. Таблиці включають таку інформацію:

- об'єм основної пам'яті, відведений процесу;
- об'єм вторинної пам'яті, відведений процесу;
- усі атрибути захисту блоків основної і віртуальної пам'яті;
- уся інформація, необхідна для управління віртуальною пам'яттю.

Детально ці інформаційні структури, які використовуються для управління пам'яттю, розглядатимемо пізніше в наступних розділах, а зараз коротко нагадаємо поняття «Віртуальної пам'яті».

Віртуальна пам'ять – це пристрій, що дозволяє програмістам розглядати пам'ять з логічної точки зору, не піклуючись про фізичну пам'ять достатнього об'єму. Принципи роботи з віртуальною пам'яттю були розроблені, щоб завдання декількох користувачів, виконуючись паралельно, могли одночасно бути присутніми в основній пам'яті. Із-за відмінностей в кількості пам'яті, що вимагається для різних процесів, при перемиканні процесора з одного процесу на інший важко компактно розмістити їх в основній пам'яті. Тому були розроблені системи із сторінковою організацією пам'яті, при якій пам'ять розбивається на блоки фіксованого розміру, що називаються ***сторінками.*** Звернення програми до слова пам'яті відбувається за **віртуальною адресою,** яка складається з номера сторінки і зміщення відносно її початку.

Сторінки одного і того ж процесу можуть бути розкидані по усій основній пам'яті. Система розбиття на сторінки забезпечує динамічну відповідність між віртуальною адресою, що використовується програмою, і реальною або фізичною адресою основної пам'яті.

Наступним логічним кроком розвитку в цьому напрямі було виключення вимоги, щоб усі сторінки процесу одночасно знаходилися в основній пам'яті. Досить, щоб усі вони зберігалися на диску. Під час виконання процесу тільки деякі його сторінки знаходяться в основній пам'яті. Якщо програма звертається до сторінки, яка там відсутня, апаратне забезпечення, що управляє пам'яттю, виявить це і організовує завантаження відсутніх сторінок. Така схема називається віртуальною пам'яттю.

Таблиця введення-виведення (I/O tables) використовується ОС для управління пристроями введення-виведення і каналами комп'ютерної системи. У кожен момент часу пристрій введення-виведення може бути або вільним, або відданий в розпорядження якогось процесу. Якщо виконується операція

введення-виведення, ОС повинна мати інформацію про її стан і про те, які адреси основної пам'яті задіяні в цій операції. Управління введенням-виведенням розглядатиметься пізніше.

Таблиці файлів. У цих таблицях знаходиться інформація про існуючі файли, їх розташування на магнітних носіях, поточний стан і інші атрибути. Велика частина цієї інформації, якщо не вся, може підтримуватися системою управління файлами. В цьому випадку ОС мало знає (чи зовсім нічого не знає) про файли. Ця тема також детально розглядатиметься в розділі «Управління файлами».

Таблиці процесів. Нарешті, ОС повинна підтримувати таблиці процесів **(блоки управління процесами**), щоб мати можливість управляти ними. Врешті-решт, управління пам'яттю, пристроями введення-виведення і файлами здійснюється для того, щоб могли виконуватися процеси, тому в таблицях процесів мають бути явні або неявні посилання на ці ресурси.

5.8.2 Структури управління процесами

Розглянемо питання про те, які відомості повинні бути в розпорядженні ОС, щоб вона могла управляти процесом. По-перше, вона повинна знати, де знаходитися процес, а по-друге, їй мають бути відомі необхідні для управління атрибути процесу (такі, як його ідентифікатор, стан і розміщення в пам'яті).

Упродовж існування процесу його виконання може бути багаторазово перерване і продовжене. Для того щоб відновити виконання процесу, необхідно відновити стан його операційного середовища. Стан операційного середовища відображається станом регістрів і програмного лічильника, режимом роботи процесора, покажчиками на відкриті файли, інформацією про незавершені операції введення-виведення тощо.

Блок управляючого процесу. Для того щоб операційна система могла виконувати операції над процесами, кожен процес представляється деякою структурою даних.

Розглянемо ще раз, в чому ж полягає фізичний прояв процесу. Як мінімум, у процес входить програма, яку треба виконати. З цією програмою пов'язаний набір елементів пам'яті, в яких зберігаються локальні і глобальні змінні. Таким чином, процесу має бути виділений такий об'єм пам'яті, в якому помістилися б програма і дані. Крім того, при роботі програми використовується стек, за допомогою якого реалізуються виклики процедур і передача параметрів. Нарешті, з кожним процесом пов'язані декілька атрибутів, які використовуються ОС для управління цим процесом. Тому, ОС для реалізації планування процесів потрібно структура, що містить інформацію, специфічну для цього процесу:

- стан, в якому знаходиться процес;
- програмний лічильник процесу або, іншими словами, адреса команди, яка має бути виконана наступною;
- вміст регістрів процесора;

– дані, необхідні для планування використання процесора і управління пам'яттю (пріоритет процесу, розмір і розташування адресного простору тощо);
– облікові дані (ідентифікаційний номер процесу, загальний час використання процесора цим процесом тощо);
– інформацію про пристрої введення-виведення, пов'язані з процесом (наприклад, які пристрої закріплені за процесом, таблицю відкритих файлів).

Такий набір атрибутів називається ***управляючим блоком процесу*** (PCB – Process Control Block). Часто використовуються інші назви цієї структури даних ***– дескриптор процесу, блок управління задачою, дескриптор задачі***.

Блок управління процесом є моделлю процесу для операційної системи. Будь-яка операція, виконана операційною системою над процесом, викликає певні зміни в PCB.

Дескриптори окремих процесів об'єднані в список, що утворює таблицю процесів. Пам'ять для таблиці процесів відводиться динамічно в області ядра. На підставі інформації, що міститься в таблиці процесів, операційна система здійснює планування і синхронізацію процесів. У дескрипторі прямо або побічно (через покажчики, на пов'язані з процесом структури) міститься інформація про стан процесу, про розташування образу процесу в оперативній пам'яті і на диску, про значення окремих складових пріоритету. Також у таблиці процесів знаходиться інформація про його підсумкові значення – глобальний пріоритет, ідентифікатор користувача процесу, споріднені процеси, події, здійснення яких чекає цей процес, тощо.

Конкретний склад PCB і будова залежать, звичайно, від конкретної операційної системи. У багатьох операційних системах інформація, що характеризує процес, зберігається не в одній, а в декількох пов'язаних структурах даних. Ці структури можуть мати різні найменування, містити додаткову інформацію або, навпаки, лише частину описаної інформації. Для нас це не має значення. Для нас важливо лише те, що для будь-якого процесу, що знаходиться в обчислювальній системі, вся інформація, необхідна для здійснення операцій над ним, доступна операційній системі.

Контекст процесу. Інформацію, для зберігання якої призначений блок управління процесом, зручно для подальшого викладу розділити на дві частини.

Вміст усіх регістрів процесора (включаючи значення програмного лічильника) називатимемо ***регістровим контекстом процесу***, а усе інше – ***системним контекстом процесу***. Знання регістрового і системного контекстів процесу вистачає для того, щоб управляти його поведінкою в операційній системі, здійснюючи над ним операції. Проте цього недостатньо, щоб повністю характеризувати процес.

Операційну систему не цікавить, якими саме обчисленнями займається процес, тобто який код і які дані знаходяться в його адресному просторі. З точки зору користувача, навпаки, найбільший інтерес представляє вміст адресного простору процесу, можливо разом з регістровим контекстом, що визначає послідовність перетворення даних і отримані результати. Код і дані, що

знаходяться в адресному просторі процесу, називатимемо його ***контекстом користувача.*** Сукупність регістрового, системного і призначеного для користувача контекстів процесу скорочено прийнято називати просто ***контекстом процесу***. У будь-який момент часу процес повністю характеризується своїм контекстом.

Множина, в яку входять програма, дані, стек і атрибути (управляючий блок процесу), називається ***образом процесу*** (process image). Місцезнаходження образу процесу залежить від використовуваної схеми управління пам'яттю. У простому випадку образ процесу має вигляд безперервного блоку пам'яті, який розташований у вторинній пам'яті, зазвичай на диску. Щоб ОС могла управляти процесом, принаймні, невелика частина його образу повинна знаходитися в основній пам'яті. Щоб запустити процес, його образ необхідно повністю завантажити в основну пам'ять.

5.8.3 Атрибути процесів

Багатозадачна система повинна мати в розпорядженні відомості про кожен процес. Різні ОС організовують цю інформацію по-різному. Тому розглянемо питання про те, яка інформація (типові елементи управляючого блоку процесу) може знадобитися ОС, не зупиняючись на схемі організації цієї інформації.

Ідентифікація процесів.

Ідентифікатори. Числові ідентифікатори, які можуть зберігатися в управляючому блоці процесу:

1. *Ідентифікатор процесу*. Зазвичай це числовий ідентифікатор. При створенні дочірніх процесів ідентифікатори вказують батьківський і дочірні процеси.
2. *Ідентифікатор батьківського процесу*.
3. *Ідентифікатор користувача*.
4. *Інформація про стан процесу*.

Регістри, доступні користувачеві. Це регістри, до яких можна звернутися за допомогою машинних команд. Зазвичай їх від 8 до 32, хоча в деяких реалізаціях RISC процесорів (з обмеженим набором команд) їх понад 100.

Управляючі регістри і регістри стану. У процесорі є декілька різновидів регістрів, які використовуються для управління роботою процесора. Перериваючи процес, усю інформацію, що міститься в регістрах, необхідно зберегти, щоб потім відновити при відновленні виконання процесу. До них належать:

1. *Лічильник команд*. У цьому регістрі зберігається адреса чергової команди.
2. *Коди умови.* Відбиває результат виконання арифметичної або логічної операції (наприклад, знак, рівність, переповнювання).
3. *Інформація про стан.* Сюди входять прапори дозволу переривань і інформація про режим виконання, слово стану процесора PSW (Processor Status Word). Цей регістр містить біти коду станів, які задаються командами порівняння, пріоритетом центрального процесора, режимом користувача/ядра, та іншу інформацію.

4. *Покажчики на стек.* З кожним процесом пов'язані один або декілька системних стеків. У стеку зберігаються параметри і адреси викликів процедур і системних служб. Покажчик стека вказує на його вершину.

Управляча інформація процесу.

Інформація про планування і стан. Зазвичай включає таке:

1. *Стан процесу*. Тобто, виконується, готовий до виконання, очікуючий якоїсь події або призупинений.
2. *Пріоритет.* У деяких ОС їх може бути декілька (за замовчуванням, поточний, максимально можливий).
3. *Інформація, що пов'язана з плануванням.* Залежить від використаного алгоритму планування.
4. *Інформація про події.* Ідентифікація події, настання якої дозволить продовжити виконання процесу, що знаходиться в очікуванні.

Структуризація даних. Процес може бути пов'язаний з іншими процесами за допомогою черги, кільця або іншої структури. Наприклад, усі процеси в стані очікування, що мають один і той же пріоритет, можуть знаходитися в одній черзі. Процеси можуть мати родинні відношення (бути батьківськими або дочірніми стосовно один до одного), тому можуть бути покажчики на процеси.

Обмін інформацією між процесами. Різні прапори, сигнали і повідомлення можуть мати стосунки до обміну інформацією між двома незалежними процесами.

Привілеї процесів. Процеси можуть мати привілеї прав доступу до певних областей пам'яті, виконувати деякі команди або використати різні системні утиліти і служби.

Управління пам'яттю. Цей розділ може містити покажчик на програмний сегмент, покажчик на сегмент даних, покажчики на таблиці сегментів і/або сторінок, в яких описується розподіл процесу у віртуальній пам'яті.

Управління файлами. Розділ може містити кореневий каталог, робочий каталог, дескриптори файлу, ідентифікатор користувача.

Володіння ресурсами і їх використання. Вказуються ресурси, якими управляє процес (наприклад, перелік відкритих файлів).

Особливо слід зауважити, що в інформації про стан процесора є опис регістра або набору регістрів, відомих під назвою «***слово стану програми***» (Program Status Word – PSW), в яких міститься інформація про стан і коди умов.

5.8.4 Управління процесами

Перш ніж обговорити метод, який ОС використовує для управління процесами, розглянемо спочатку розбіжності між режимами роботи процесора при виконанні коду ОС, і при виконанні кодів програм користувача. Більшість процесорів підтримують два режими роботи. Певні команди виконуються тільки в більше привілейованому режимі. До них належать команди читання або внесення змін до управляючих регістрів (LDTR, GDTR та ін.), команди введення-

виведення, а також команди, які пов'язані з управлінням пам'яттю. Крім того, доступ до деяких ділянок пам'яті може бути дозволений тільки в привілейованому режимі.

Режим з меншими привілеями називають ***режимом користувача***, оскільки в ньому виконуються програми користувача. Режим з вищими привілеями називається ***системним режимом*** (system mode), ***режимом управління*** (control mode) або ***режимом ядра*** (kernel mode). Ядро – це частина ОС, яка виконує найважливіші її функції, серед яких виділимо наведені нижче.

1. Управління процесами:

- створення і завершення процесів;
- планування і диспетчеризація процесів;
- перемикання процесів;
- синхронізація і підтримка обміну інформацією між процесами;
- організація управляючих блоків процесів.

2. Управління пам'яттю:

- виділення адресного простору процесам;
- свопінг;
- управління сторінками і сегментами;
- управління введенням-виведенням;
- управління буферами;
- виділення процесам каналів і пристроїв введення-виведення.

3. Функції підтримки:

- обробка переривань;
- облік використання ресурсів;
- поточний контроль системи.

З наведеного легко зрозуміти, для чого потрібні два вище згаданих режими. Це необхідно для захисту ОС і її основних структур, таких як управляючі блоки процесів, від можливого впливу програм користувачів. Програми, які працюють в режимі ядра, мають повний контроль над процесором і всіма його командами і регістрами, а також мають доступ до всіх елементів пам'яті. Такий рівень привілеїв для програм користувачів не потрібний, тому, виходячи з міркувань безпеки, його роблять недоступним для програм користувача.

У зв'язку з цим виникає два питання: яким чином процесор може визначити, в якому режимі може виконуватися ця програма, і як здійснюється перемикання з одного режиму в інший? Відносно першого питання, то в PSW програми є спеціальний біт (VM), де вказаний режим виконання. При деяких подіях цей біт змінюється. Наприклад, якщо програма викличе сервіс ОС, встановлюється режим ядра. Це виникає в результаті виконання команд зміни режиму. Якщо програма користувача спробує виконати таку команду, то це призведе до передачі управління ОС, а якщо така зміна режиму не дозволена, то виникне помилка виконання.

5.9 Операції над процесами

5.9.1 Набір операцій

Процес сам не може перейти з одного стану в інший. Зміною стану процесів займається операційна система, здійснюючи операції над ними. Кількість таких операцій в нашій моделі співпадає з кількістю стрілок на діаграмі станів. Їх зручно об'єднати в три пари:

1. Створення процесу – завершення процесу.
2. Призупинення процесу (перехід із стану ***виконання*** в стан ***готовність)*** – запуск процесу (перехід із стану ***готовність*** в стан ***виконання***).
3. Блокування процесу (перехід із стану ***виконання*** в стан ***очікування)*** – розблокування процесу (перехід із стану ***очікування*** в стан ***готовність***).

Надалі, коли ми вивчатимемо алгоритми планування, в нашій моделі з'явиться ще одна операція, що не має пари: зміна пріоритету процесу.

Операції створення і завершення процесу є одноразовими, оскільки застосовуються до процесу не більше одного разу (деякі системні процеси ніколи не завершуються при роботі обчислювальної системи). Усі інші операції, пов'язані зі зміною стану процесів, будь то запуск або блокування, як правило, є багаторазовими. Розглянемо детальніше, як операційна система виконує операції над процесами.

5.9.2 Одноразові операції

Складний життєвий шлях процесу в комп'ютері розпочинається з його народження. Будь-яка операційна система, що підтримує концепцію процесів, повинна мати засоби для їх створення. У дуже простих системах (наприклад, в системах, спроектованих для роботи тільки для одного конкретного застосування) усі процеси можуть бути породжені на етапі старту системи. Складніші ОС створюють процеси динамічно, в міру необхідності. Ініціатором народження нового процесу після старту операційної системи може виступити або процес користувача, що вчинив спеціальний системний виклик, або сама операційна система, тобто, зрештою, теж деякий процес. Процес, що ініціював створення нового процесу, прийнято називати ***процесом-батьком*** (parent process), а знову створений процес – ***процесом-дитиною*** (child process). Процеси-діти можуть, у свою чергу, породжувати нові процеси тощо.

При народженні процесу система заводить новий PCB із станом процесу ***народження*** і починає його заповнення. Новий процес отримує свій власний унікальний ідентифікаційний номер.

Для виконання своїх функцій процес-дитина вимагає певних ресурсів: пам'яті, файлів, пристроїв введення-виведення тощо. Існує два підходи до їх виділення. Новий процес може отримати у своє користування деяку частину батьківських ресурсів, можливо, розділяючи з процесом-батьком і іншими процесами-дітьми права на них, або може отримати свої ресурси безпосередньо від операційної системи. Інформація про виділені ресурси заноситься в PCB.

Після наділу процесу-дитини ресурсами необхідно занести в його адресний простір програмний код, значення даних, встановити програмний лічильник. Тут також можливі два рішення. У першому випадку процес-дитина стає дублікатом процесу-батька за регістровим і призначеним для користувача контекстам. При цьому повинен існувати спосіб визначення хто для кого з процесів-двійників є батьком. У другому випадку процес-дитина завантажується новою програмою з якого-небудь файлу.

Операційна система UNIX дозволяє породження процесу тільки першим способом; для запуску нової програми необхідно спочатку створити копію процесу-батька, а потім процес-дитина повинна замінити свій призначений для користувача контекст за допомогою спеціального системного виклику. Операційні системи VAX/VMS і WINDOWS NT допускають тільки друге рішення.

Породження нового процесу як дубліката процесу-батька призводить до можливості існування програм (тобто виконуваних файлів), для роботи яких організовується більше чим один процес. Можливість заміни призначеного для користувача контексту процесу по ходу його роботи (тобто, завантаження для виконання нової програми) призводить до того, що в рамках одного і того ж процесу можуть бути послідовно виконані декілька різних програм.

Після того як процес наділений змістом, в PCB дописується інформація, що залишилася, і стан нового процесу змінюється на ***готовність***.

Після завершення процесу, операційна система переводить його в стан ***закінчив виконання*** і звільняє усі асоційовані з ним ресурси, роблячи відповідні записи в блоці управління процесом. При цьому сам PCB не знищується, а залишається в системі ще деякий час. Це пов'язано з тим, що процес-батько після завершення процесу-дитини може запросити операційну систему про причину завершення процесу і/або статистичну інформацію про його роботу. Подібна інформація зберігається в PCB «мертвого» процесу до запиту процесу-батька або до кінця його діяльності, після чого всі сліди процесу остаточно зникають з системи. В операційній системі UNIX процеси, що знаходяться в стані ***закінчив виконання***, прийнято називати процесами ***зомбі.***

5.9.3 Багаторазові операції

Одноразові операції призводять до зміни кількості процесів, що знаходяться під управлінням операційної системи, і завжди пов'язані з виділенням або звільненням певних ресурсів. Багаторазові операції, навпаки, не призводять до зміни кількості процесів в операційній системі і не пов'язані з виділенням або звільненням ресурсів.

Запуск процесу. З числа процесів, що знаходяться в стані *готовність*, операційна система вибирає один процес для подальшого виконання. Критерії і алгоритми такого вибору будуть детально розглянуті в розділі «Планування процесів». Для обраного процесу операційна система забезпечує наявність в оперативній пам'яті інформації, необхідної для його подальшого виконання. Те, як вона це робить, буде в деталях описано в розділі «Управління пам'яттю». Далі

стан процесу змінюється на виконання, відновлюються значення регістрів для цього процесу, і управління передається команді, на яку вказує лічильник команд процесу. Всі дані, необхідні для цього відновлення контексту, витягаються з PCB процесу, над яким здійснюється операція.

Призупинення процесу. Робота процесу, що знаходиться в стані *виконання*, призупиняється в результаті якого-небудь переривання. Процесор автоматично зберігає лічильник команд і, можливо, один або декілька регістрів у стеку виконуваного процесу і передає управління за спеціальною адресою обробки цього переривання. На цьому діяльність апаратної частини з обробки переривання завершується. За вказаною адресою розташовується одна з частин ОС. Вона зберігає динамічну частину системного і регістрового контекстів процесу в його PCB, переводить процес в стан *готовність* і приступає до обробки переривання.

Блокування процесу. Процес блокується, коли він не може продовжувати свою роботу, не дочекавшись виникнення якої-небудь події в обчислювальній системі. Для цього він звертається до операційної системи за допомогою певного системного виклику. Операційна система обробляє системний виклик (ініціалізує операцію введення-виведення, додає процес в чергу процесів, що чекають звільнення пристрою або виникнення події тощо). ОС зберігає необхідну частину контексту процесу в його PCB і переводить процес із стану *виконання* в стан *очікування*. Детальніше ця операція розглядатиметься в розділі «Управління введенням-виведенням».

Розблокування процесу. Після виникнення в системі якої-небудь події, ОС треба точно визначити яка саме подія сталося. Потім операційна система перевіряє: чи знаходився деякий процес у стані *очікування* для цієї події і, якщо знаходився, переводить його в стан *готовність*, виконуючи необхідні дії, пов'язані з настанням події (ініціалізація операції введення-виведення для чергового очікуючого процесу тощо). Ця операція, як і операція блокування, буде детальніше описана в розділі «Управління введенням-виведенням».

5.9.4 Перемикання контексту

Досі ми розглядали операції над процесами ізольовано, незалежно одна від одної. Насправді ж діяльність мультипрограмної операційної системи складається з ланцюжків операцій, що виконуються над різними процесами і супроводжуються перемиканням процесора з одного процесу на інший. Для прикладу спрощено розглянемо, як в реальності може виникати операція розблокування процесу, очікуючого введення-виведення (рис. 5.8).

При виконанні процесором деякого процесу (на рисунку – процес 1) виникає переривання від пристрою введення-виведення, який сигналізує про закінчення операцій на пристрої. Над процесом, що виконується, робиться операція призупинення. Далі, операційна система розблоковує процес, що ініціював запит на уведення-виведення (на рис. 5.8 – процес 2), і здійснює запуск призупиненого або нового процесу, вибраного при виконанні планування (на рис. 5.8 був вибраний розблокований процес).

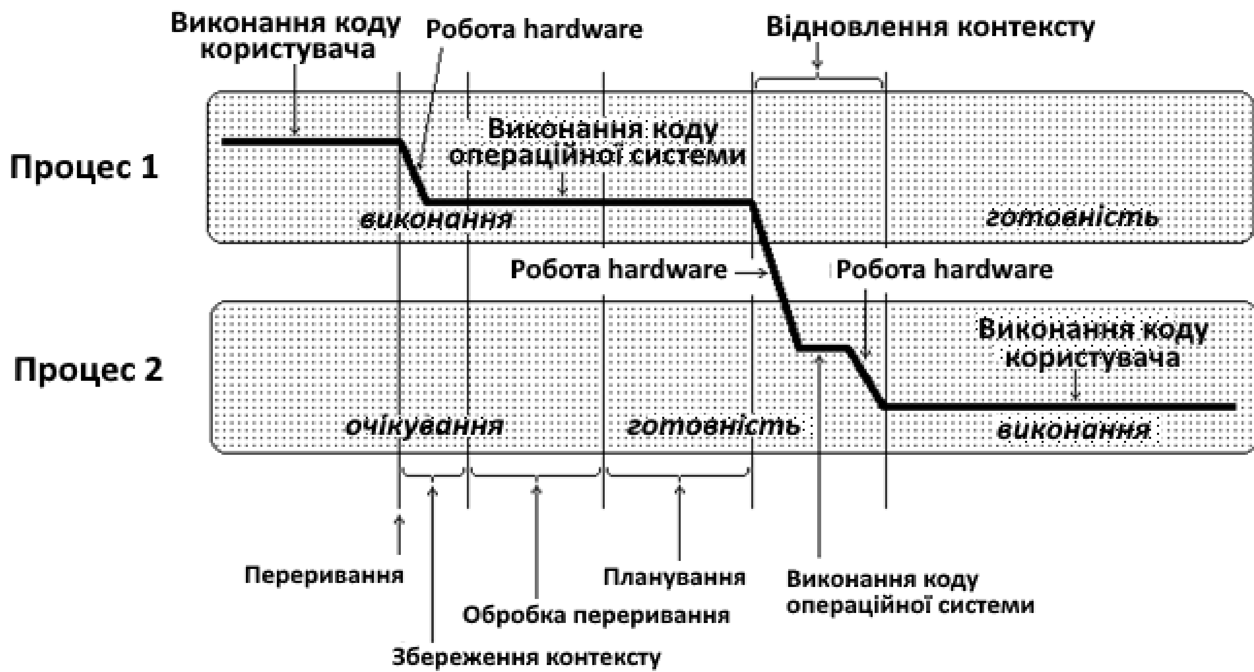


Рисунок 5.8 – Виконання операції розблокування процесу

Для коректного перемикання процесора з одного процесу на інший необхідно зберегти контекст процесу, що виконувався, і відновити контекст процесу, на який буде перемкнутий процесор. Така процедура збереження/відновлення працездатності процесів називається ***перемиканням контексту***. Час, витрачений на перемикання контексту, не використовується обчислювальною системою для здійснення корисної роботи і є накладними витратами, що знижують продуктивність системи. Він змінюється від машини до машини і зазвичай знаходиться в діапазоні від 1 до 1000 мікросекунд.

5.10 Взаємодія процесів

У багатозадачному і мережевому середовищі процеси повинні якось спілкуватися один з одним. У більшості ОС передбачені механізми взаємодії процесів, які зобов'язані координувати (синхронізувати) свої зусилля для досягнення загального результату. Ситуації, коли процесам доводиться взаємодіяти:

1. Передача інформації від одного процесу іншому.
2. Контроль над діяльністю процесів. Наприклад, коли вони борються за один ресурс.
3. Узгодження дій процесів. Наприклад, коли один процес поставляє дані, а інший їх виводить на друк. Якщо узгодженості не буде, то другий процес може почати друк раніше, ніж поступлять дані.

Другий і третій випадок стосуються і потоків, які ми розглядатимемо пізніше. У першому випадку в потоків немає ніяких труднощів, оскільки вони використовують загальний адресний простір.

Передача може здійснюватися декількома способами: пам'ять, що розділяється, сигнали, повідомлення, виклик віддаленої процедури, сокети.

5.10.1 Пам'ять, що розділяється

Традиційно вважається, що основним способом міжпроцесного обміну є ресурс, що розділяється, такий як пам'ять, яка грунтується на відповідних об'єктах ядра. При цьому виникають задачі створення, іменування і захисту таких ресурсів. Один з процесів створює ресурс, що розділяється, наділяє його атрибутами захисту і ім'ям, за яким цей ресурс може бути доступний іншим процесам (навіть у разі завершення роботи процесу-творця). Як приклад розглянемо спілкування через пам'ять, що розділяється (рис. 5.9).

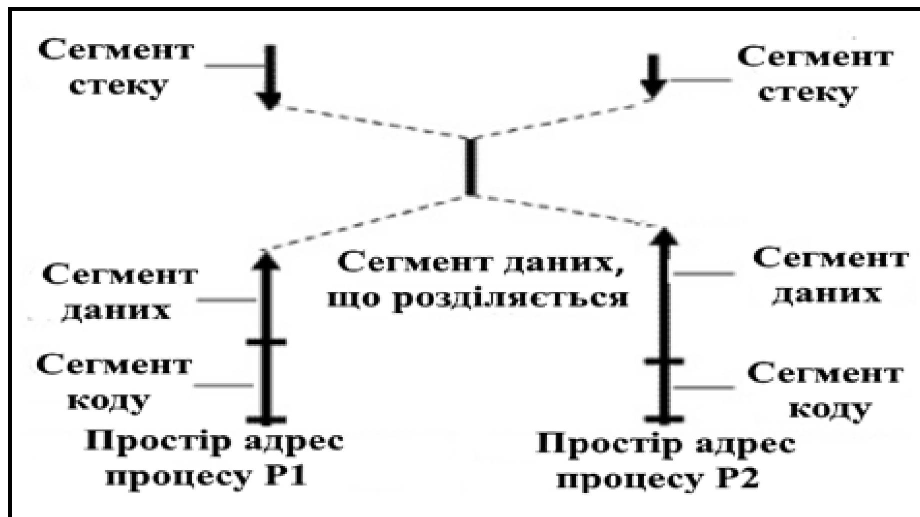


Рисунок 5.9 – Адресні простори процесів, взаємодіючих через сегмент пам'яті

Наприклад, в ОС Windows сегмент пам'яті, що розділяється, створюється за допомогою Win32-функції *CreateFileMapping*. Це фрагмент пам'яті, доступний за іменем (параметр *lpname*), який базується на відповідному об'єкті ядра. Процесу-творцеві повертається описувач (*handle*) ресурсу. Інші процеси, що бажають мати доступ до ресурсу, також повинні отримати його описувач. В даному випадку це можна зробити за допомогою функції *OpenFileMapping*, вказавши ім'я ресурсу в якості одного з параметрів [28].

5.10.2 Сигнали

Сигналами називають програмні переривання, що повідомляють процес про настання певної події. На відміну від інших механізмів взаємодії процесів, сигнали не дозволяють процесам обмінюватися один з одним якою-небудь інформацією. Системні сигнали залежать від операційної системи і типів програмних переривань, підтримуваних певним процесором. При надходженні сигналу ОС спочатку визначає, кому призначений цей сигнал, а потім – як процес повинен на нього відреагувати. Процеси можуть перехоплювати сигнали, ігнорувати їх або маскувати.

Процес перехоплює сигнал і визначає процедуру, що викликається ОС у разі надходження сигналу.

Процеси можуть проігнорувати сигнал, тобто перекласти відповідальність за виконання дії по обробці сигналу за умовчанням на операційну систему.

Найчастіше за умовчанням задається аварійне завершення (abort) процесу. Іншим варіантом дії за умовчанням є ігнорування сигналу.

Процеси можуть заблокувати обробку сигналу шляхом його ***маскування***. Коли процес маскує сигнал певного типу (наприклад, сигнал призупинення), операційна система блокує сигнали цього типу, поки маскування сигналу не буде відключено.

5.10.3 Передача повідомлень

З розвитком розподілених систем зріс інтерес до взаємодії процесів за допомогою повідомлень. Повідомлення можуть передаватися в одному напрямі, тоді для будь-якого повідомлення один процес є відправником, а інший – одержувачем. Передача повідомлень може також бути двонаправленою, тобто кожен процес під час взаємодії одночасно може бути відправником і одержувачем.

Прийом і відправка повідомлень реалізується у вигляді виклику системних функцій, доступних у більшості мов програмування. У разі блокуючої передачі процес змушений чекати до тих пір, поки повідомлення не буде доставлено одержувачеві, вимагаючи підтвердження прийому.

При неблокуючій передачі процес-відправник може продовжувати виконання інших операцій, навіть якщо повідомлення ще не було доставлене одержувачеві (або він не повідомив про це відправника). Щоб реалізувати неблокуючу передачу, необхідно використати механізм буферизації повідомлень для зберігання повідомлень до моменту їх доставки одержувачеві.

Блокуюча передача є прикладом ***синхронного зв'язку***, тоді як неблокуюча передача **–** ***асинхронного***. Під час відправки повідомлення можна вказати процес-одержувач або опустити ім'я процесу, у такому разі буде зроблена ***широкомовна передача повідомлень*** усім процесам системи (***поштові скриньки*** в Windows).

Асинхронний зв'язок у поєднанні з неблокуючою передачею сприяє збільшенню швидкодії системи за рахунок зменшення часу очікування різних подій процесами. Наприклад, якщо процес відправить інформацію на зайнятий сервер друку, система зберігатиме цю інформацію до тих пір, поки сервер друку не буде готовий її прийняти, тоді як процес-відправник зможе продовжити виконання інших завдань, не чекаючи звільнення сервера друку.

Популярною реалізацією механізму передачі повідомлень є ***канал*** (труба, pipe) – захищена ОС область пам'яті, яка виступає буфером псевдофайлу, що дозволяє декільком процесам обмінюватися між собою даними. Операційна система синхронізує доступ до буфера. Після того, як записуючий процес закінчить вести запис у буфер (ймовірно заповнивши його), система призупинить роботу записуючого процесу, дозволивши процесу-читачу почати читання даних з буфера. У міру зчитування даних з буфера (ймовірно спустошивши його), операційна система, у свою чергу, призупинить його виконання, дозволивши записуючому процесу знову почати запис інформації в буфер.

Обговорюючи взаємодію процесів, що виконуються на одному комп'ютері, ми припускаємо, що обмін інформацією відбувається без помилок. У розподілених системах при передачі даних можуть виникати помилки, що в ряді випадків призводять до втрати інформації. Тому відправники і одержувачі часто взаємодіють між собою за допомогою протоколу квитування (установка перемикача в положення, що відповідає отриманому сигналу), використовуваного для підтвердження факту прийому інформації. Механізм тайм-ауту застосовується для обмеження часу очікування повідомлення про доставку. Якщо сигнал про доставку повідомлення не поступить після закінчення заданого інтервалу, повідомлення буде відправлене повторно.

Система передачі повідомлень з функцією повторної передачі даних дозволяє ідентифікувати нові повідомлення за їх порядковим номером. Одержувач повинен перевіряти ці номери, щоб знати, чи всі повідомлення були доставлені, і при необхідності розставити їх у правильному порядку. Якщо підтвердження про приймання повідомлення загубиться, і відправник вирішить передати повідомлення повторно, новому повідомленню привласнюється той же самий порядковий номер, який належав утраченому повідомленню.

5.10.4 Віддалений виклик процедур

Віддалений виклик процедур грунтується на тому, що процес, який виконується на одному комп'ютері, запускає процес на віддаленому комп'ютері. Фактично здійснюється виклик процедури, яка реально знаходиться і підтримується на іншому комп'ютері. Віддалений виклик зовні дуже схожий на локальний. Дії з передачі параметрів і отримання результату істотно відрізняються – все виконується за допомогою передач інформаційних пакетів мережею. Детальніше ці дії виглядають так, як описано нижче.

1. Програма-клієнт робить локальний виклик процедури, що називається «заглушкою», при цьому клієнтові здається, що, викликаючи «заглушку», він насправді викликає процедуру сервера. Насправді, ззадача «заглушки» – прийняти аргументи, що адресуються процедурі, перетворити їх в деякий формат і сформувати мережевий запит.
2. Мережевий запит пересилається мережею на віддалену систему, при цьому, наприклад, використовується стек протоколів TCP/IP.
3. На сервері все відбувається в зворотному порядку: «заглушка» сервера чекає на запит і при його отриманні витягає з нього параметри.
4. «Заглушка» сервера виконує виклик справжньої процедури (якій адресований запит клієнта), передаючи їй потрібні параметри.
5. Після виконання процедури при передачі результатів її роботи управління знову повертається в «заглушку» сервера, яка формує повідомлення-відгук.
6. Відгук передається клієнтові.
7. Відгук приймається «заглушкою» клієнта, яка витягає необхідні дані і передає їх програмі. Процес завершений.

5.10.5 Сокети

Сокети (англ. socket – поглиблення, гніздо, роз'єм) – підтримуваний ядром механізм програмного інтерфейсу для забезпечення обміну даними між процесами, що приховує особливості середовища і дозволяє однаково взаємодіяти процесам як на одному комп'ютері, так і в мережі (рис. 5.10).

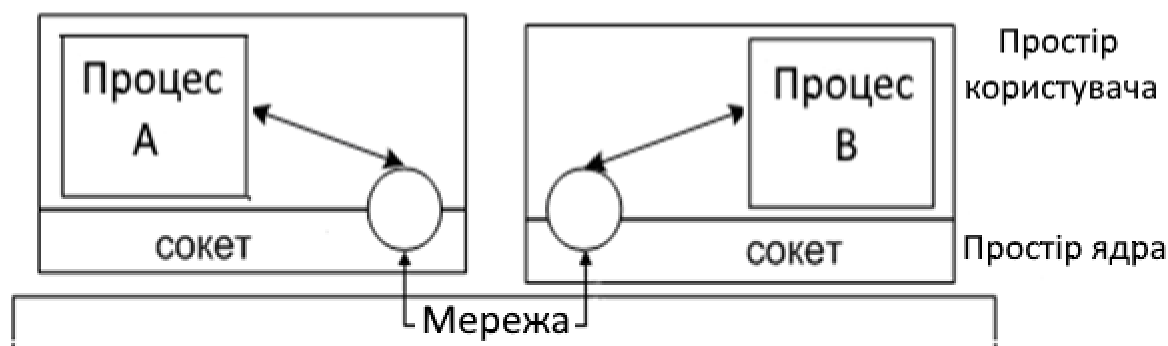


Рисунок 5.10 – Використання сокетів для роботи мережі

Слід відрізняти клієнтські і серверні сокети. Клієнтські сокети грубо можна порівняти з апаратами телефонної мережі, а серверні – з комутаторами. Клієнтськiй додаток (наприклад, браузер) використовує тільки клієнтські сокети, а серверний (наприклад, веб-сервер, якому браузер посилає запити) – як клієнтські, так і серверні сокети.

Сокети можуть динамічно створюватися і знищуватися. При створенні сокета процесу повертається дескриптор файлу для встановлення з'єднання, читання і запису даних, а також розриву з'єднання.

Інтерфейс сокетів уперше з'явився в BSD Unix. Програмний інтерфейс сокетів описаний в стандарті POSIX.1 і в тій чи іншій мірі підтримується всіма сучасними операційними системами.

Контрольні питання і тести до розділу 5

Контрольні питання

1. Які два типи одиниць роботи визначені в більшості операційних систем?
2. Чи правда, що терміни «процес» і «програма» є синонімами?
3. У якій ОС вперше почали застосовувати термін «процес»?
4. Дайте визначення процесу.
5. Який взаємозв'язок існує між завданнями, процесами і потоками?
6. Що таке вивантажний і невивантажний ресурс?
7. Яка інформація повинна бути в розпорядженні ОС для розв'язання задачі про поточний стан кожного процесу і ресурсу?
8. За яких умов ресурс може бути виділений процесу?
9. Наведіть приклади описувачів процесу в різних ОС.
10. Які основні події призводять до створення процесів?
11. Через виникнення яких подій процес закінчує свою роботу?

12. Назвіть три основні стани процесу і нарисуйте діаграму станів процесу.
13. Навіщо в ОС було введено додатковий стан призупиненого (блокованого) процесу?
14. Якщо в основній пам'яті немає жодного готового до виконання процесу, то які процеси ОС записує на диск (здійснює його свопінг)?
15. Назвіть чотири основні стани процесу при появі нового стану – призупинений.
16. Навіщо ОС використовує управляючі структури: «таблиці пам'яті», «таблиці введення-виведення» і «таблиці файлів»?
17. Яка управляюча структура потрібна ОС для реалізації планування процесів?
18. Назвіть синоніми для структури управляючого процесу.
19. Яка інформація зберігається в контексті процесу?
20. Назвіть основні операції над процесом.
21. Як називається процедура збереження (відновлення) працездатності процесів?
22. У яких ситуаціях відбувається взаємодія процесів?
23. Як взаємодіють процеси через сегмент пам'яті, що розділяється?

Тести

1. Ідентифікатор процесу є частиною **. . .** процесу.
    1) контексту;
    2) дескриптора;
    3) описувача;
    4) типу.
2. До невивантажених належать такі ресурси, які не можуть бути:
    1) відібрані у процесу;
    2) завантажені в пам'ять;
    3) вивантажені з пам'яті;
    4) відібрані у процесора.
3. Дескриптор процесу – це:
    1) стан регістрів процесора в ході виконання інструкцій;
    2) інформація, необхідна для вирішення задачі планування;
    3) інформація про відкриті файли і засоби синхронізації;
    4) початкова адреса прикладної програми в оперативній пам'яті.
4. З якого стану процес може перейти в стан «очікування»?
    1) із стану «виконання»;
    2) із стану «готовність»;
    3) із стану «народження»;
    4) із стану «блокування».
5. У який стан переводиться процес у результаті операції блокування?
    1) завршення;
    2) готовність;
    3) виконання;
    4) очікування.

6. З якого стану процес може перейти в стан «виконання»?
   1) із стану «виконання»;
   2) із стану «готовність»;
   3) із стану «народження»;
   4) із стану «блокування».
7. Як називається інформаційна структура, яка містить інформацію, необхідну для відновлення виконання процесу після переривання?
   1) процес;
   2) дескриптор;
   3) потік;
   4) контекст.
8. Який стан не визначений для процеса в системі?
   1) очікування;
   2) виконання;
   3) синхронізація;
   4) готовність.
9. Яку інформацію не містить дескриптор процесу?
   1) ідентифікатор процесу;
   2) режим роботи процесора;
   3) інформацію про стан процесу;
   4) дані про споріднені процеси.
10. Яких змін станів процесу не існує в системі?
    1) виконання → готовність;
    2) очікування → готовність;
    3) очікування → виконання;
    4) готовність → виконання.
11. Вкажіть який вид інформації з перерахованої нижче не міститься в контексті процесу:
    1) вміст регістрів процесора доступних користувачеві;
    2) покажчики на ресурси, якими управляє процес;
    3) вміст лічильника команд;
    4) стан управляючих регістрів і регістрів стану.
12. У якій з перерахованих нижче подій процес переходить в стан «заблокований»:
    1) вхід в систему;
    2) вичерпання кванта часу для виконання;
    3) очікування події;
    4) завершення виконання.
13. Який механізм може бути використаний для передачі даних від одного процесу до іншого процесу, якщо процеси виконуються на різних комп'ютерах, зв'язаних комп'ютерною мережею:
    1) канал;
    2) поштова скринька;
    3) виклик локальних процедур;
    4) сокет.

6 УПРАВЛІННЯ ПОТОКАМИ

У системах, в яких відсутнє поняття потоку, виникають проблеми при організації паралельних обчислень у рамках процесу. А така необхідність може виникати. Свого часу впровадження ідеї мультипрограмування дозволило підвищити пропускну спроможність комп'ютерних систем, тобто зменшити середній час очікування результатів роботи процесів. Але всякий розподіл ресурсів тільки уповільнює роботу одного з учасників за рахунок додаткових витрат часу на очікування звільнення ресурсу. Проте додаток, що виконується в рамках одного процесу, може мати внутрішній паралелізм, який в принципі міг би дозволити прискорити його роботу.

6.1 Концепції потоку

Потоки виникли в операційних системах як засіб розпаралелювання обчислень. Звичайно, задача розпаралелювання обчислень у рамках одного додатку може бути вирішене і традиційними способами, як це ми бачили на наведеному вище прикладі.

По-перше, прикладний програміст може взяти на себе складну задачу організації паралелізму, виділивши в додатку деяку підпрограму-диспетчер, яка періодично передає управління тієї ж або іншій гілки обчислень. При цьому програма виходить логічно дуже заплутаною, з численними передачами управління, що істотно ускладнює її відладку і модифікацію.

По-друге, використання стандартних засобів ОС для створення процесів не дозволяє врахувати той факт, що ці процеси розв'язують єдину задачу, тобто мають багато спільного між собою. Вони можуть працювати з одними і тими ж даними, використовувати один і той же кодовий сегмент, наділятися одними і тими ж правами доступу до ресурсів обчислювальної системи. Так, якщо в прикладі з сервером баз даних створювати окремі процеси для кожного запиту, що поступає з мережі, то усі процеси виконуватимуть один і той же програмний код і виконуватимуть пошук в записах, загальних для всіх процесів файлів даних. А операційна система при такому підході розглядатиме ці процеси нарівні з усіма іншими процесами і за допомогою універсальних механізмів забезпечуватиме їх ізоляцію один від одного.

У даному випадку всі ці досить громіздкі механізми використовуються явно не за призначенням, виконуючи не лише даремну, але і шкідливу роботу, що утрудняє обмін даними між різними частинами додатку. Крім того, на створення кожного процесу ОС витрачає певні системні ресурси, які в даному випадку невиправдано дублюються, – кожному процесу виділяються власний віртуальний адресний простір, фізична пам'ять, за ним закріплюються пристрої введення-виведення тощо.

Якщо, наприклад, у програмі передбачено звернення до зовнішнього пристрою, то на час цієї операції можна не блокувати виконання всього процесу, а продовжити обчислення по іншій гілці програми. Паралельне виконання декількох робіт у рамках одного інтерактивного додатку підвищує ефективність роботи користувача. Так, при роботі з текстовим редактором бажано мати

можливість поєднувати набір нового тексту з такими тривалими за часом операціями, як переформатування значної частини тексту, друк документу або його збереження на локальному або віддаленому диску.

Ще одним прикладом необхідності розпаралелювання є мережевий сервер баз даних. У цьому випадку паралелізм бажаний як для обслуговування різних запитів до бази даних, так і для швидшого виконання окремого запиту за рахунок одночасного перегляду різних записів бази даних.

Розглянемо простий приклад. Нехай у нас є така програма на псевдомові програмування.

Ввести масив A
Ввести масив B
Ввести масив C
A = A + B; C = A + C
Вивести масив C

При виконанні такої програми в рамках одного процесу цей процес чотири рази блокуватиметься, чекаючи закінчення операцій введення-виведення. Але наш алгоритм має внутрішній паралелізм. Обчислення суми масивів *A + B* можна було б виконувати паралельно з очікуванням закінчення операції введення масиву *C*. Таке поєднання операцій за часом можна було б реалізувати, використовуючи два взаємодіючі процеси. Для простоти вважатимемо, що засобом комунікації між ними служить пам'ять, що розділяється. Тоді наші процеси можуть виглядати таким чином.

Процес 1	**Процес 2**
Ввести масив A	*Очікування введення масивів A і B*
Очікування закінчення операції введення	
Ввести масив B	
Очікування закінчення операції введення	
Ввести масив C	
Очікування закінчення операції введення	*A = A + B*
C = A + C	
Вивести масив C	
Очікування закінчення операції виведення	

Здавалося б, ми запропонували конкретний спосіб прискорення розв'язання задачі. Проте, насправді, справа йде не так просто. Другий процес має бути створений, обидва процеси повинні повідомити операційну систему, що їм потрібна пам'ять, яку вони могли б розділити з іншим процесом, і, нарешті, не можна забувати про перемикання контексту. Тому реальна поведінка процесів виглядатиме приблизно так.

Процес 1		**Процес 2**
Створити процес 2		
	Перемикання контексту	
		Виділення загальної пам'яті
		Очікування введення A і B
	Перемикання контексту	

Виділення загальної пам'яті
Ввести масив А
Очікування закінчення операції введення
Ввести масив В
Очікування закінчення операції введення
Ввести масив С
Очікування закінчення операції введення

Перемикання контексту

А = А + В

Перемикання контексту

С= А + С
Вивести масив
Очікування закінчення операції виведення

Очевидно, що ми можемо не лише не виграти в часі при розв'язанні задачі, але навіть і програти, оскільки часові втрати на створення процесу, виділення загальної пам'яті і перемикання контексту можуть перевищити виграш, отриманий за рахунок поєднання операцій.

З усього вищевикладеного витікає, що в ОС, разом з більшою одиницею роботи (процесами), потрібний інший механізм розпаралелювання обчислень, який враховував би тісні зв'язки між окремими гілками обчислень одного і того ж джодатку, і вимагав для свого виконання дещо дрібніших робіт. Для цих цілей сучасні ОС пропонують механізм багатопотокової обробки (multithreading). При цьому вводиться нова одиниця роботи – ***потік виконання*** [9], а поняття «процес» значною мірою міняє сенс. Поняттю «потік» відповідає послідовний перехід процесора від однієї команди програми до іншої. ОС розподіляє процесорний час між потоками. ОС призначає процесу адресний простір і набір ресурсів, які спільно використовуються усіма його потоками.

Говорячи про процеси, ми відмічали, що ОС підтримує їх відособленість: у кожного процесу є свій віртуальний адресний простір, кожному процесу призначаються свої ресурси – файли, вікна тощо. Така відособленість потрібна для того, щоб захистити один процес від іншого, оскільки вони, спільно використовуючи усі ресурси машини, конкурують один з одним. У загальному випадку процеси належать різним користувачам, що розділяють один комп'ютер, і ОС бере на себе роль арбітра в спорах процесів за ресурси.

Проте задача, що вирішується в рамках одного процесу, може мати внутрішній паралелізм, який, в принципі, дозволяє прискорити його розв'язання. Наприклад, в ході виконання задачі відбувається звернення до зовнішнього пристрою, і на час цієї операції можна не блокувати повністю виконання процесу, а продовжити обчислення по іншій «гілці» процесу. Таким чином, для реалізації «мультизадачності» в її істинному тлумаченні необхідно теж ввести відповідну суть. Такою суттю і стали так звані ***процеси «легкої ваги» (полегшені процеси),*** або ***міні-процеси,*** або, як їх тепер називають, **– *потоки*** або ***треди*** (***нитки***, thread).

Створення потоків вимагає від ОС менших накладних витрат, ніж процесів. На відміну від процесів, які належать різним конкуруючим додаткам, усі потоки одного процесу завжди належать одному додатку, тому ОС ізолює потоки в набагато меншому ступені, ніж процеси в традиційній мультипрограмній системі.

Усі потоки одного процесу використовують загальні файли, таймери, пристрої, одну і ту ж область оперативної пам'яті, один і той же адресний простір. Це означає, що вони розділяють одні і ті ж глобальні змінні. Оскільки кожен потік може мати доступ до будь-якої віртуальної адреси процесу, один потік може використати стек іншого потоку. Між потоками одного процесу немає повного захисту, оскількі, по-перше, це неможливо, а, по-друге, неепотрібно. Щоб організувати взаємодію і обмін даними, потокам зовсім не потрібно звертатися до ОС, їм достатньо використати загальну пам'ять – один потік записує дані, а інший їх читає.

Другим аргументом на користь потоків є легкість (тобто швидкість) їх створення і ліквідації в порівнянні з «важкими» процесами. У багатьох системах створення потоків здійснюється в 10-100 разів швидше, ніж створення процесів. Ця властивість особливо згодиться, коли потрібно буде швидко і динамічно змінювати кількість потоків.

Отже, мультипрограмування ефективніше на рівні потоків, а не процесів. Кожен потік має власний лічильник команд і стек. Задача, яка оформлена у вигляді декількох потоків у рамках одного процесу, може бути виконане швидше за рахунок псевдопаралельного (чи паралельного в мультипроцесорній системі) виконання її окремих частин. Наприклад, якщо електронна таблиця була розроблена з урахуванням можливостей багатопотокової обробки, то користувач може запросити перерахунок свого робочого листа і одночасно продовжувати заповнювати таблицю. Особливо ефективно можна використати багатопоточність для виконання розподілених додатків, наприклад, багатопотоковий сервер може паралельно виконувати запити відразу декількох клієнтів.

Використання потоків пов'язане не лише з прагненням підвищити продуктивність системи за рахунок паралельних обчислень, але і з метою створення читабельніших, логічніших програм. Наприклад, в задачах типу «письменник-читач» один потік виконує запис у буфер, а інший прочитує записи з нього. Оскільки вони ділять між собою загальний буфер, то не потрібно їх робити окремими процесами. Інший приклад використання потоків – управління сигналами, такими як переривання з клавіатури (del або break). Замість обробки сигналу переривання один потік призначається для постійного очікування сигналів. Таким чином, використання потоків може скоротити необхідність в перериваннях рівня користувача. У цих прикладах не таке важливе паралельне виконання, наскільки важлива ясність програми.

Найбільший ефект від введення багатопотокової обробки досягається в мультипроцесорних системах, в яких потоки, у тому числі і ті, що належать одному процесу, можуть виконуватися на різних процесорах дійсно паралельно (а не псевдопараллельно).

6.2 Модель потоку

У багатопотоковій системі при створенні процесу ОС створює для кожного процесу адресний простір і як мінімум один потік виконання. Фактично це майже визначення процесу. При створенні потоку так само, як і при створенні процесу, операційна система генерує спеціальну інформаційну структуру – ***описувач потоку***, який містить ідентифікатор потоку, дані про права доступу і пріоритет, стан потоку і іншу інформацію. У початковому стані потік (чи процес, якщо йдеться про систему, в якій поняття «потік» не визначається) знаходиться в призупиненому стані. Момент вибірки потоку на виконання здійснюється відповідно до прийнятого в цій системі правила надання процесорного часу і з урахуванням усіх існуючих на даний момент потоків і процесів. У випадку, якщо коди і дані процесу знаходяться в області підкачування, необхідною умовою активізації потоку процесу є також наявність місця в оперативній пам'яті для завантаження його виконуваного модуля.

Коли говорять про процеси, то тим самим хочуть відмітити, що у кожного процесу є свій адресний простір, що містить текст програми і дані, кожному процесу призначаються свої ресурси. Іншими словами, у разі процесів ОС вважає їх абсолютно незв'язаними і незалежними. При цьому ОС бере на себе роль арбітра в конкуренції між процесами з приводу ресурсів.

Концепція потоку додає до моделі процесу можливість одночасного виконання в одному і тому ж середовищі процесу декількох програм, достатньою мірою незалежних, Декілька потоків, працюючих паралельно в одному процесі, аналогічні декільком процесам, що йдуть паралельно на одному комп'ютері. Потоки мають деякі властивості процесів, тому їх іноді називають ***спрощеними процесами.*** Термін ***багатопоточність*** також використовують для опису використання декількох потоків в одному процесі.

На рис. 6.1, *а* представлені три звичайні процеси, у кожного з яких є власний адресний простір і один потік управління. На рис. 6.1, *б* представлено один процес з трьома потоками управління.

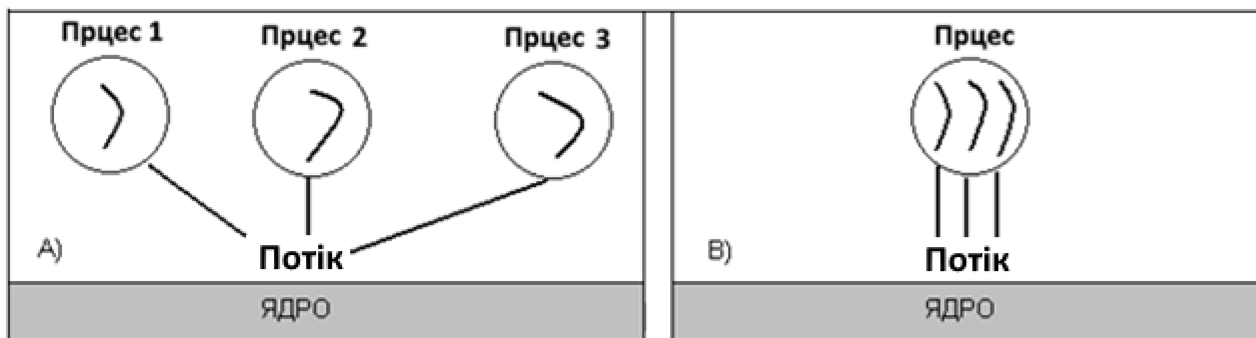


Рисунок 6.1 – Три процеса з поодинокими потоками (А); один процес з трьома потоками (В)

При запуску багатопотокового процесу в системі з одним процесором потоки працюють по черзі. Ілюзія паралельної роботи декількох різних послідовних процесів створюється шляхом постійного перемикання системи між

процесами. Багатопоточність реалізується приблизно так же. Процесор швидко перемикається між потоками, створюючи враження паралельної роботи потоків.

Різні потоки в одному процесі не так незалежні, як різні процеси. В усіх потоків один і той же адресний простір, що означає спільне використання глобальних змінних. Оскільки будь-який потік має доступ до будь-якої адреси елементу пам'яті процесу, один потік може прочитувати, записувати інформацію в стек іншого потоку. Захисту не існує, оскільки це неможливо і не потрібно. Як показано в табл. 6.1. потоки розділяють не лише адресний простір, але й відкриті файли, дочірні процеси тощо.

Таблиця 6.1 – Елементи процесів и потоків

Елементи процесу	Елементи потоків
Адресний простір Глобальні змінні Відкриті файли Дочірні процеси Необроблені аварійні сигнали Сигнали і їх обробники Інформація про використання ресурсів	Лічильник команд Регістри Стек Стан

У 1-ій колонці перераховані елементи, спільно використовувані усіма потоками процесу, а в 2-ій – елементи індивідуальні для кожного потоку.

Перша колонка містить елементи, що є властивостями процесу, а не потоку. Наприклад, якщо процес відкриває файл, цей файл тут же стає видимим для потоків, і вони можуть прочитувати і записувати інформацію у файл. Це логічно, оскільки процес, а не потік є одиницею управління ресурсами. Якби у кожного потоку був власний адресний простір і інші ресурси, то це були б окремі процеси.

Легковаговими ці задачі називають ще тому, що ОС не повинна для них організовувати повноцінну віртуальну машину. Ці задачі не мають своїх власних ресурсів, вони розвиваються в тому ж адресному просторі, можуть користуватися тими ж файлами, віртуальними пристроями і іншими ресурсами, що і цей процес. Єдине, що їм необхідно мати, – це процесорний ресурс. В однопроцесорній системі потоки розділяють між собою процесорний час так само, як це роблять звичайні процеси, а в мультипроцесорній системі можуть виконуватися одночасно.

Слід врахувати, що кожен потік має свій власний стек, що і показано на рис. 6.2. Стек кожного потоку містить по одному фрейму для кожної вже викликаної, але такої, що ще не повернула управління, процедури. Такий фрейм містить локальні змінні процедури і адресу повернення управління після закінчення її виклику. Наприклад, якщо процедура *X* викликає процедуру *Y*, а *Y* викликає процедуру *Z*, то при виконанні *Z* в стеку будуть фрейми для *X, Y* і *Z.* Кожен потік буде, як правило, викликати різні процедури і, отже, мати середовище виконання, що відрізняється від інших. Тому кожному потоку потрібний свій власний стек.

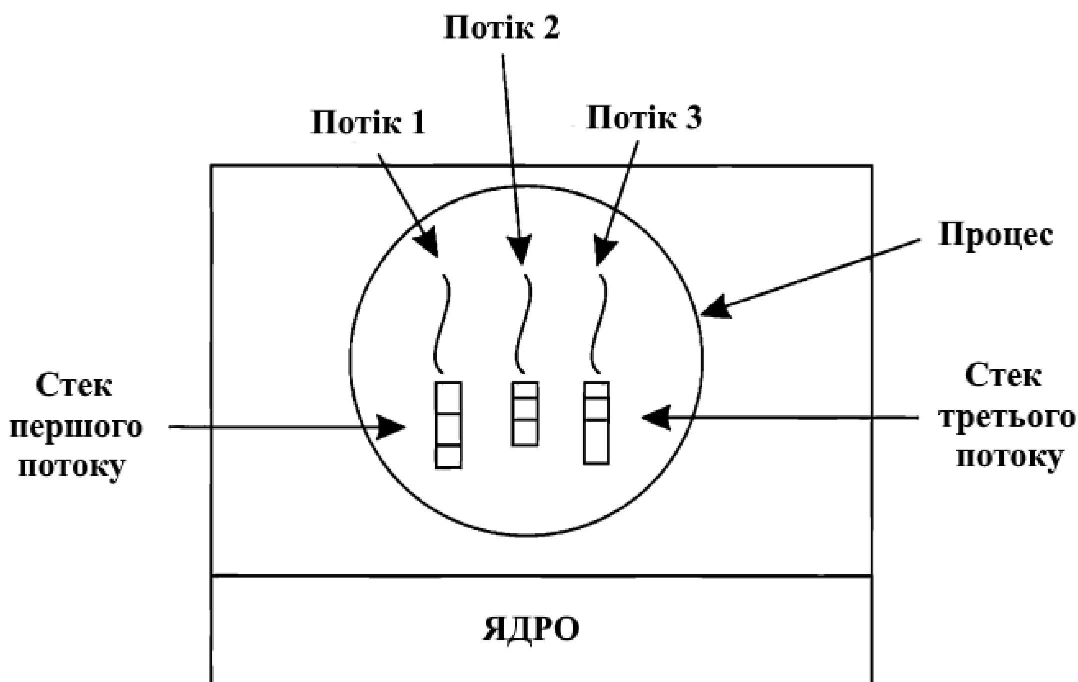


Рисунок 6.2 – Для кожного потоку є свій власний стек

6.3 Використання потоків

Настав час пояснити, чому ж потоки такі потрібні. Необхідність потоків простіше продемонструвати на конкретному прикладі. Уявіть собі, що користувач пише книгу. З точки зору автора найпростіше зберігати книгу в одному файлі, щоб легше було шукати окремі розділи, виконувати глобальну заміну і тому подібне. З іншого боку, можна зберігати кожну главу в окремому файлі.

Тепер уявіть собі, що користувач захотів видалити на 1-ій сторінці документу, в якому 800 сторінок, одне речення, а потім вирішив виправити речення на 600 сторінці (наприклад, задавши пошук фрази, що зустрічається тільки на цій сторінці). Текстовому редакторові доведеться переформатувати увесь документ аж до 600 сторінки. Це може зайняти відносно багато часу.

У цьому випадку допоможуть потоки. Нехай редактор написаний у вигляді двохпотокової програми. Один потік взаємодіє з користувачем, а другий переформатовує документ у фоновому режимі. Як тільки речення на першій сторінці була видалено, інтерактивний потік дає команду фоновому потоку переформатувати увесь документ. У тей час коли перший потік продовжує відстежувати і виконувати команди з клавіатури або миші, другий потік швидко переформатовує документ. Форматування може закінчитися раніше, ніж користувач захоче перейти до 600 сторінки, і тоді команда буде виконана миттєво.

Чому б у даному прикладі не додати ще один потік? Більшість текстових редакторів зберігають редагований текст один раз в декілька хвилин. Цим може займатися третій потік, не відволікаючи два, що залишилися.

Якби програма була однопоточною, тоді при кожній операції форматування або збереження файлу всі команди з клавіатури і миші

ігнорувалися б до закінчення цих операцій. У користувача це створило б враження низької продуктивності. Програмний модуль з трьома потоками істотно простіший. Очевидно, що в цьому випадку модель з трьома процесами не буде невдалою, оскільки всім трьом необхідно працювати з одним і тим же документом. Три ж потоки спільно використовують загальну пам'ять, і все три мають доступ до документу.

Паралельні обчислення (а, отже, і ефективніше використання ресурсів центрального процесора, і менший сумарний час виконання задач) тепер уже часто реалізується на рівні потоків. Програма, яка оформлена у вигляді декількох потоків у рамках одного процесу, може бути виконана швидше за рахунок паралельного виконання її окремих частин. Наприклад, якщо електронна таблиця розроблена з урахуванням можливостей багатопотокової обробки, то користувач може запросити перерахунок робочого листа і одночасно продовжувати заповнювати таблицю або редагувати наступний документ.

Потреба в потоках виникла ще на однопроцесорних системах, оскільки вони дозволяють організувати обчислення ефективніше. Для багатопроцесорних систем потоки вже просто потрібні, оскільки вони дозволяють не лише реально прискорити виконання тих задач, які допускають їх природне розпаралелювання, але і завантажити процесор роботою, щоб він не простоював.

Унаслідок того, що потоки належать до одного процесу, виконуються в одному віртуальному адресному просторі, між ними легко організувати тісну взаємодію, на відміну від процесів, для яких потрібні спеціальні механізми обміну повідомленнями і даними. Ще одним аргументом на користь потоків є легкість їх створення і знищення, оскільки з потоком не пов'язані ніякі ресурси. На створення потоку йде приблизно в 100 разів менше часу, ніж на створення процесу.

Незважаючи на перераховані переваги потоків, використання потоків не є універсальним засобом розв'язання проблем паралелізму, і пов'язано з деякими утрудненнями.

1. Розробляти і відлагоджувати багатопотокові програми важче, ніж звичайні послідовні програми. Організація загального використання адресного простору декількома потоками вимагає, щоб програміст мав високу кваліфікацію.
2. Використання потоків може понизити продуктивність програми. Найчастіше це трапляється в однопроцесорних системах. Наприклад, спроба виконати складні розрахунки декількома потоками може зробити до зайвих витрат на перемикання між потоками, кількість виконуваних інструкцій залишається одна і та ж.

Переваги і недоліки використання потоків необхідно враховувати під час виконання будь-якого проекту. Наведемо декілька порад з використання потоків.

1. У разі використання однопроцесорної системи певна кількість паралельних потоків часто не прискорює роботу додатку, якщо кожен з потоків не вимагатиме частого введення-виведення, оскільки буде тільки додаткове навантаження на систему, витрачене на перемикання між потоками.

2. Потоки добре виконуються, коли вони незалежні. Але вони починають працювати непродуктивно, якщо вони змушені часто синхронізуватися для доступу до загальних ресурсів. Блокування і критичні секції не збільшують швидкість роботи системи.

3. Пам'ять віртуальна. Механізм віртуальної пам'яті стежить за тим, яка частина віртуального адресного простору повинна знаходитися в оперативній пам'яті, а яка має бути скинута у файл підкачування. Потоки ускладнюють ситуацію, якщо вони звертаються в один і той же час до різних адрес віртуального адресного простору додатка. Це значно збільшує навантаження на систему, особливо при невеликому об'ємі кеш-пам'яті.

4. Не слід покладати на потік декілька функцій. Чим простіше і менш багатозначна кожна з даних ситуацій, тим більше ймовірность того, що помилок вдасться уникнути.

5. Всякий раз, коли який-небудь з потоків намагається скористатися загальним ресурсом цього процесу, якому він належить, потрібно тим або іншим чином легалізувати і захистити свою діяльність. Хорошим засобом для цього є критичні секції, семафори і черги повідомлень.

Кожен потік виконується строго послідовно і має свій власний програмний лічильник і стек. Подібно до традиційних процесів (тобто процесів, що складаються з одного потоку), потоки можуть також породжувати ***потоки-нащадки***, можуть переходити із стану в стан (виконання, очікування і готовність). Поки один потік заблокований, інший потік того ж процесу може виконуватися. Потоки ділять між собою процесор так, як це роблять процеси, відповідно до різних варіантів планування. Поки один потік заблокований (чи просто знаходиться в черзі готових до виконання задач), інший потік того ж процесу може виконуватися.

У різних ОС по-різному будуються стосунки між потоками-нащадками і їх батьками. Наприклад, в одних ОС виконання батьківського потоку синхронізується з його нащадками, зокрема після завершення батьківського потоку ОС може знімати з виконання усіх його нащадків. В інших системах потоки-нащадки можуть виконуватися асинхронно стосовно батьківського потоку.

6.4 Стани потоку

Подібно до традиційних процесів, потоки можуть також переходити із стану в стан. Вважається, що процес знаходиться в стані **готовність**, якщо хоч би один з його потоків знаходиться в стані **готовність** і жоден з потоків не знаходиться в стані **виконання.** Також вважається, що процес знаходиться в станіі **виконання**, якщо один з його потоків знаходиться в стані **виконання.** Процес знаходитиметься в стані **очікування**, якщо усі його потоки знаходяться в стані **очікування.** Поки один потік процесу заблокований, інший потік того ж процесу може виконуватися. Нарешті, процес знаходиться в стані **закінчив виконання**, якщо всі його потоки знаходяться в стані **закінчив виконання.**

Потоки розділяють процесор так само, як це робили традиційні процеси, відповідно до розглянутих алгоритмів планування.

ОС виконує планування потоків, зважаючи на їх стан. У мультипрограмній системі потік може знаходитися в одному з трьох основних станів:

- ***виконання*** – активний стан потоку, під час якого потік має усі необхідні ресурси і безпосередньо виконується процесором;
- ***очікування*** – пасивний стан потоку, знаходячись в якому потік заблокований зі своїх внутрішніх причин (чекає здійснення деякої події, наприклад, завершення операції введення-виведення, отримання повідомлення від іншого потоку або звільнення якого-небудь необхідного йому ресурсу);
- ***готовність*** – також пасивний стан потоку, але в цьому випадку потік заблокований в зв'язку із зовнішньою стосовно нього обставиною (має усі потрібні для нього ресурси, готовий виконуватися, проте процесор зайнятий виконанням іншого потоку).

Слід зауважити, що стани виконання і очікування можуть бути віднесені і до задач, що виконуються в однопрограмному режимі, а ось стан готовності характерний тільки для режиму мультипрограмування.

Впродовж свого життя кожен потік переходить з одного стану в інший відповідно до алгоритму планування потоків, прийнятого в цій операційній системі.

Оскільки потоки одного процесу розділяють істотно більше ресурсів, ніж різні процеси, то операції створення нового потоку і перемикання контексту між потокми одного процесу займають значно менше часу, ніж аналогічні операції для процесів у цілому.

Розглянемо типовий граф стану потоку (рис. 6.3).

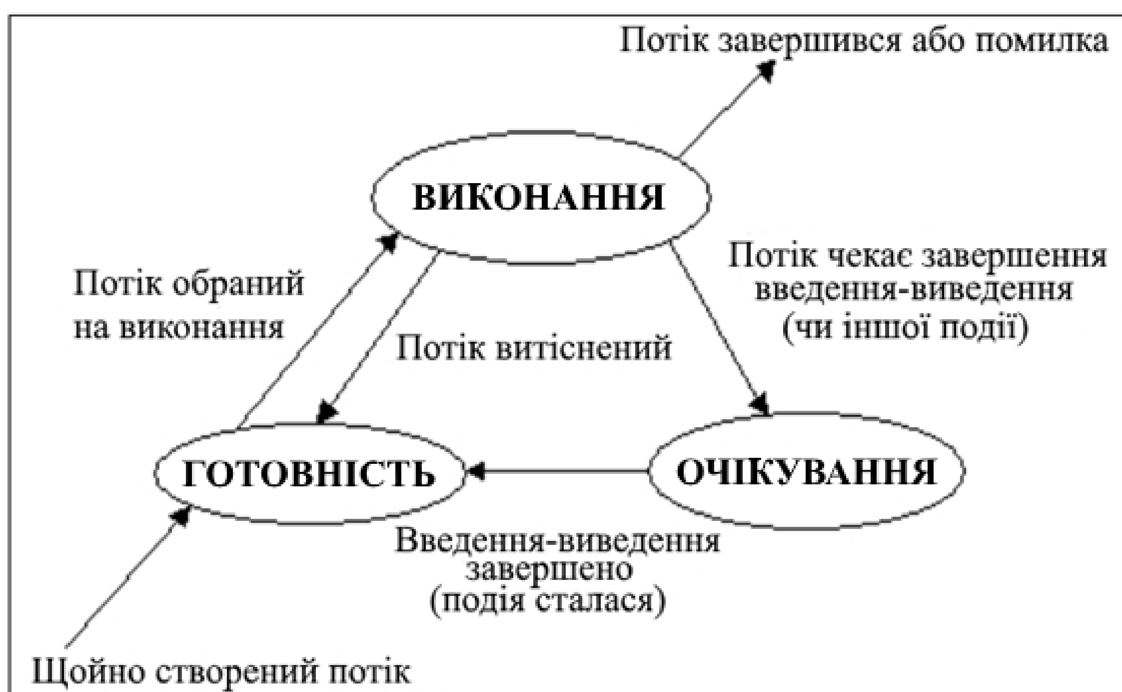


Рисунок 6.3 – Стани потоку в багатозадачному середовищі

Тільки що створений потік знаходиться в стані готовності, він готовий до виконання і стоїть в черзі до процесора. Коли в результаті планування підсистема управління потоками приймає рішення про активізацію цього потоку, він переходить у стан виконання і знаходиться в ньому до тих пір, поки він сам звільнить процесор, перейшовши в стан очікування якої-небудь події, або буде примусово «витіснений» з процесора, наприклад, внаслідок вичерпання відведеного цьому потоку кванту процесорного часу. В останньому випадку потік повертається в стан готовності. У цей же стан потік переходить із стану очікування, після того, як очікувана подія станеться.

У стані виконання в однопроцесорній системі може знаходитися не більше ніж один потік, а в кожному із станів очікування і готовності – декілька потоків. Ці потоки утворюють черги відповідно очікуючих і готових потоків. Черги потоків організовуються шляхом об'єднання в списки описувачів окремих потоків. Таким чином, кожен описувач потоку, крім усього іншого, утримує, принаймні, один покажчик на інший описувач, що знаходиться поряд з ним в черзі. Така організація черг дозволяє легко їх переупорядковувати, включати і виключати потоки, переводити потоки з одного стану в інший. Якщо припустити, що на рис. 6.4 показана черга готових потоків, то запланований порядок виконання виглядає так: A, B, E, D, C.

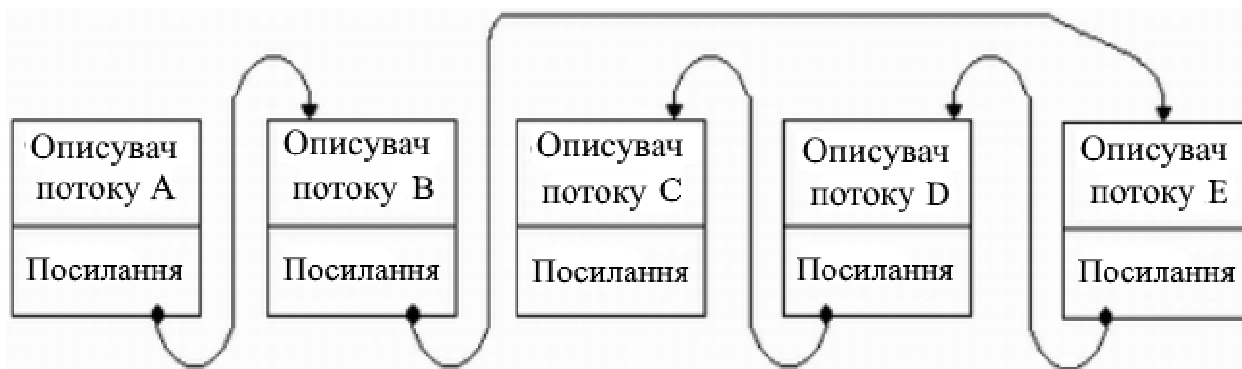


Рисунок 6.4 – Черга потоків

6.5 Потоки на рівні користувача і на рівні ядра

Виділяють дві загальні категорії потоків: потоки на рівні користувача (User-Level Threads – ULT) і потоки на рівні ядра (Kernel-Level Threads – KLT). Потоки другого типу називаються потоками, підтримуваними ядром або ***полегшеними (легковагими) процесами***.

6.5.1 Потоки на рівні користувача (ULT)

У програмі, яка складається повністю з ULT-потоків, усі дії з управління потоками виконуються самими додатком, ядро і не підозрює про існування потоків. На рис. 6.5 показаний підхід, коли використовується тільки потоки на рівні (у просторі) користувача. Щоб додаток був багатопотоковим його потрібно створити з використанням спеціальної бібліотеки (система підтримки програм), яка є пакетом програм для роботи з потоками на рівні ядра.

Така бібліотека містить код, який дозволяє створювати і видаляти потоки, здійснювати обмін повідомленнями і даними між потоками, планувати їх виконання, а також зберігати і відновлювати їх контекст.

За умовчанням додаток на початку своєї роботи складається з одного потоку, і його виконання починається як виконання цього потоку. Такий додаток разом з його потоком розміщується в єдиному процесі, який управляється ядром.

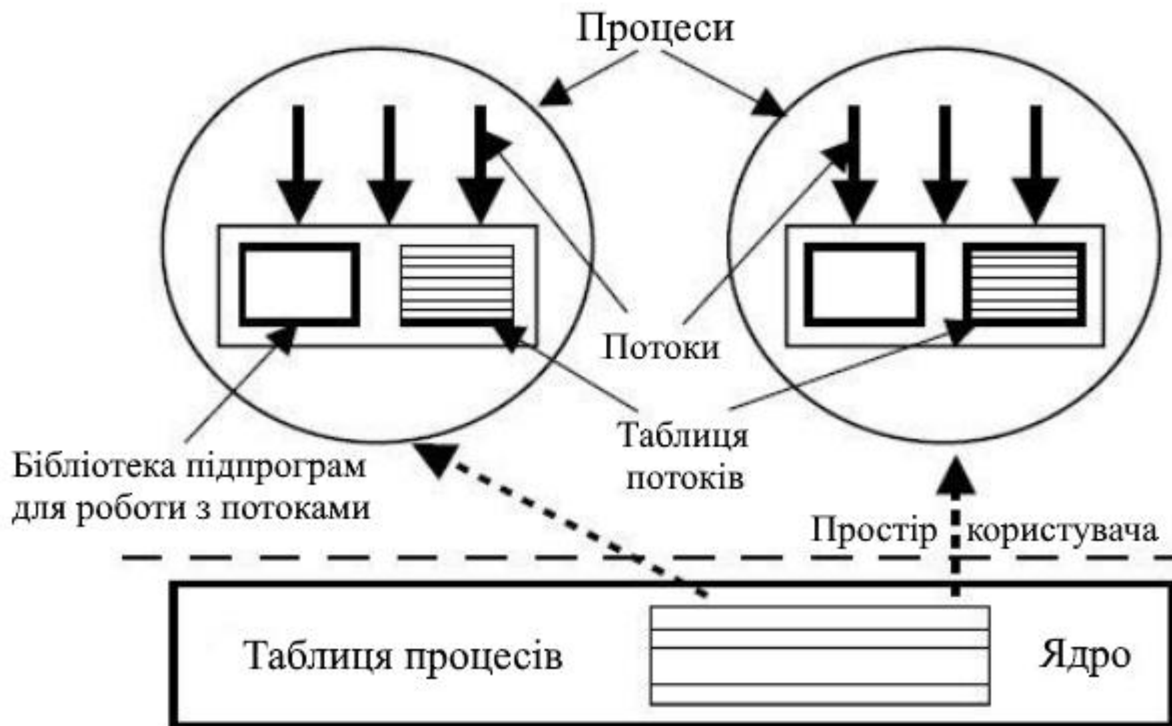


Рисунок 6.5 – Потоки в простоорі користувача

Додаток, що виконується, у будь-який момент часу може створити новий потік, який виконуватиметься в межах того ж процесу. Цей новий потік створюється за допомогою виклику спеціальної підпрограми з бібліотеки, яка призначена для роботи з потоками. Управління до цієї підпрограми переходить у результаті виклику процедури. Бібліотека потоків створює структуру даних для нового потоку, а потім передає управління одному з готових до виконання потоків цього процесу відповідно до деякого алгоритму планування.

Коли управління переходить до бібліотечної підпрограми (система підтримки виконання програм), контекст поточного потоку зберігається, а коли управління повертається до потоку, його контекст відновлюється. Цей контекст (таблиця потоків), необхідний для відстежування потоків у процесі, в основному складається з вмісту регістрів користувача, лічильника команд і покажчиків стека. Кожному процесу потрібна ***таблиця потоків*** для відстежування потоків у процесі. Вона аналогічна таблиці процесів, з тією лише різницею, що вона відстежує лише характеристики потоків, такі як лічильник команд, покажчик вершини стека, регістри, стан тощо.

При цьому, усі описані раніше події відбуваються в призначеному для користувача просторі в рамках одного процесу. Ядро не підозрює про існування

потоків. Воно продовжує здійснювати планування процесу як єдиного цілого і приписувати йому єдиний стан виконання (стан готовності, стан виконання, стан блокування тощо).

Використання потоків на призначеному для користувача рівні має деякі переваги перед використанням потоків на рівні ядра.

1. Перемикання потоків не включає перехід в режим ядра, оскільки структури даних по управлінню потоками знаходяться в адресному просторі одного і того ж процесу. Тому для управління потоками процесу не треба перемикатися в режим ядра. Завдяки цій обставині вдається уникнути накладних витрат, які пов'язані з двома перемиканнями режимів (режиму користувача в режим ядра і назад).
2. Планування здійснюється залежно від специфіки додатку. Для одних додатків кращим може бути простий алгоритм планування за круговим алгоритмом, а для інших – алгоритм планування на використанні пріоритету.
3. Використання потоків на призначеному для користувача рівні застосовне для будь-якої ОС. Для їх підтримки в ядро системи не потрібно вносити ніяких змін. Бібліотека потоків є набором утиліт, що працюють на рівні додатку і спільно використовуються усіма додатками.

Використання потоків на призначеному для користувача рівні має два явні недоліки в порівнянні з використанням потоків на рівні ядра:

1. У типовій ОС більшість системних викликів є блокуючими. Коли в потоці, який працює в режимі користувача, виконується системний виклик, то блокується не лише цей потік, але й усі потоки того процесу, до якого він належить.
2. У стратегії з наявністю потоків тільки на призначеному для користувача рівні додаток не може скористатися перевагами багатопроцесорної системи, оскільки ядро закріплює за кожним процесом тільки один процесор. Тому декілька потоків одного і того ж процесу не можуть виконуватися одночасно. По суті, у нас виходить багатозадачність на рівні додатку в рамках одного процесу.

Ці дві проблеми вирішувані. Наприклад, їх можна здолати, якщо писати додаток не у вигляді декількох потоків, а у вигляді декількох процесів. Але при такому підході основні переваги потоків зводяться нанівець: кожне перемикання стає не перемиканням потоків, а перемиканням процесів, що призводить до значних накладних витрат.

Іншим методом подолання проблеми блокування є використання перетворення блокуючого системного виклику в неблокуючий. Наприклад, замість безпосереднього виклику системної процедури введення-виведення потік викличе програмну оболонку, яка здійснює уведення-виведення на рівні додатку. У цій програмі міститься код, який перевіряє, чи зайнятий пристрій введення-виведення. Якщо він зайнятий, то потік передає управління іншому потоку (за допомогою бібліотеки потоків). Коли потік знову отримує управління, він повторно здійснює перевірку зайнятості пристрою введення-виведення. Але подолання усіх цих проблем виливається в складність їх реалізації.

6.5.2 Потоки на рівні ядра (KLT)

Тепер розглянемо ситуацію, в якій ядро знає про існування потоків і управляє ними. У програмах, робота яких повністю заснована на потоках, що працюють на рівні ядра, усі дії з управління потоками виконуються ядром. В області додатків відсутній код, який призначений для управління потоками. Немає необхідності і в наявності таблиці потоків у кожному процесі, замість цього є єдина таблиця потоків, що відстежує всі потоки системи. Замість коду управління потоками в процесі використовуються інтерфейс прикладного програмування (API) засобів ядра, що управляють потоками. Прикладами такого підходу є ОС OS/2, LINUX і W2K.

На рис. 6.6 показана стратегія використання потоків на рівні ядра. Будь-який додаток при цьому можна запрограмувати як багатопотоковий; усі потоки додатка підтримуються в рамках єдиного процесу. Ядро підтримує інформацію контексту процесу як єдиного цілого, а також контекстів кожного окремого потоку процесу.

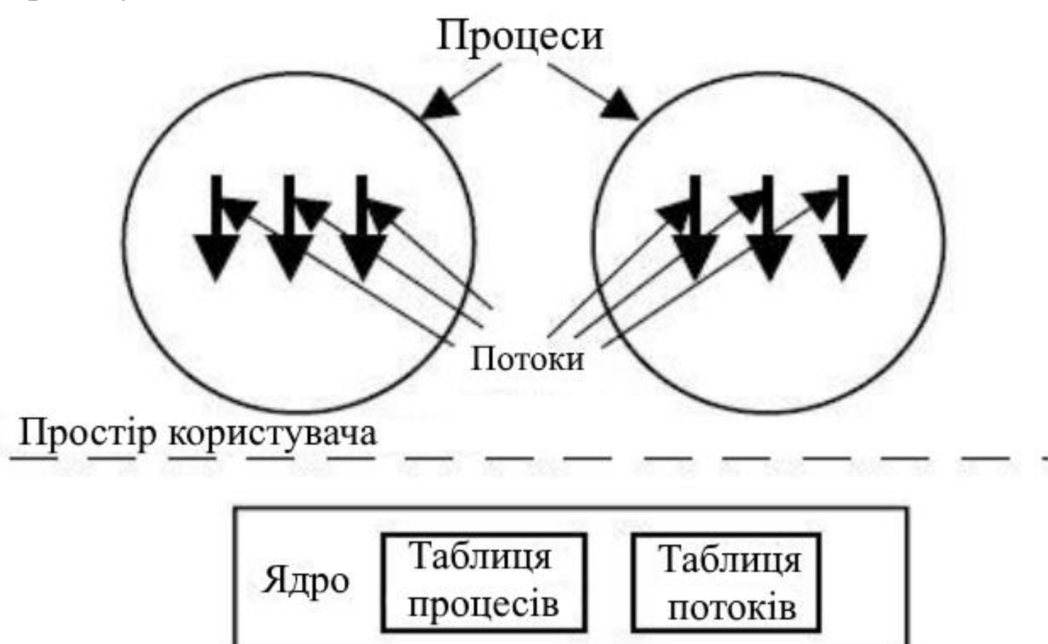


Рисунок 6.6 – Потоки в просторі ядра

Планування виконується ядром виходячи зі стану потоків. За допомогою такого підходу вдається позбавитися від двох згаданих раніше недоліків потоків рівня користувача.

По-перше, ядро може одночасно здійснювати планування роботи декількох потоків одного і того ж процесу на декількох процесорах. По-друге, при блокуванні одного із потоків процесу ядро може вибрати для виконання інший потік цього ж процесу. Ще однією перевагою такого підходу є те, що самі процедури ядра можуть бути багатопоточними.

Основним недоліком підходу з використанням потоків на рівні ядра, в порівнянні з використанням потоків на рівні користувача, є те, що для передачі управління від одного потоку до іншого в рамках одного і того ж процесу доводитися перемикатися в режим ядра.

6.5.3 Комбіновані підходи

У деяких ОС використовується комбінування потоків обох видів (Solaris).

Декілька потоків на рівні користувача, що входять до складу додатку, відображаються в таку ж або меншу кількість потоків на рівні ядра. Програміст може змінювати число потоків на рівні ядра, підбираючи його таким, яке дозволяє досягти найкращих результатів.

При комбінованому підході кілька потоків одного і того ж додатку можуть одночасно виконуватися на декількох процесорах, а блокуючі системні виклики не призведуть до блокування всього процесу. При належній реалізації такий підхід буде поєднувати в собі переваги підходів, в яких використовується тільки потоки на рівні користувача або тільки потоки на рівні ядра, зводячи недоліки кожного з цих підходів до мінімуму.

6.5.4 Спливаючі потоки

Потоки часто використовуються в розподілених системах. Комп'ютери в розподілених системах взаємодіють, обмінюючись один з одним повідомленнями. Важливим прикладом може служити обробка вхідних повідомлень, наприклад запитів на обслуговування. Традиційний підхід полягає в наявності процесу або потоку, який блокується за системним запитом, чекаючи повідомлення. Коли повідомлення прибуває, воно приймається і обробляється.

Можливий і інший підхід, при якому після прибуття повідомлення система створює новий потік для його обробки, який називають ***спливаючим***.

Основною перевагою спливаючих потоків є їх «свіжість» (вони створюються заново) – у такого потоку немає історії: регістрів, стека і іншої інформації, яку потрібно відновлювати.

Створення спливаючих потоків у просторі ядра завжди швидше і простіше, ніж у просторі користувача. До того ж зі спливаючого потоку в просторі ядра простіше отримати доступ до всіх таблиць ядра і пристроїв введення-виведення. З іншого боку, наявність помилок в потоці, розташованому в просторі ядра, може завдати істотно більший збиток.

6.6 Багатопотокове програмування

Ефективне використання апаратури, що підтримує паралелізм, і в першу чергу так званих SMP систем (Symmetric Multiprocessor Architectures – **SMP;** симетричної мультипроцесорної архітектури комп'ютерів), тобто машин, що мають декілька процесорів і загальну, тобто доступну кожному з них пам'ять, дозволило ширше впроваджувати механізм **легковагових процесів**, або потоків управління (threads).

На подібній апаратурі програма, що використовує потоки може виконувати більш ніж одну інструкцію одночасно. Це дозволить додатку основну частину часу роботи, яку займають обчислення, підвищити свою продуктивність на двопроцесорній машині практично в два рази.

Перш ніж перейти до конкретних переваг, які дає нам багатопотоквое програмування, необхідно пояснити значення деяких термінів які ми активно використовуватимемо.

Під ***асинхронними*** подіями ми розумітимемо події, які відбуваються незалежно (можливо одночасно) за винятком випадків, коли залежність встановлюється зовнішніми силами.

Терміном ***конкуренція*** описуватимемо ситуацію, коли здається, що процеси відбуваються одночасно, проте насправді вони можуть відбуватися послідовно. Цим терміном добре описується, наприклад, поведінка процесів, що одночасно виконуються на однопроцесорній машині, хоча нам і здається, що зараз виконується декілька процесів, але насправді в цей конкретний момент виконується тільки один процес, що отримав поточний квант процесорного часу.

Терміном ***паралелізм*** описуватимемо ситуацію, коли два процеси виконуються одночасно, тобто, паралельно, не перетинаючись. Справжній паралелізм може проявлятися тільки на багатопроцесорних системах, тоді як ***конкуренція*** і на однопроцесорних і на багатопроцесорних системах. Іншими словами конкуренція є лише ілюзією паралелізму. А справжній паралелізм вимагає для одночасного виконання декількох процесів декількох виконавців.

Використовуючи потоки і конкуренцію, ми можемо підвищити продуктивність програм навіть без використання апаратури, що забезпечує істинний паралелізм.

Припустимо, що ми маємо два потоки в програмі, що виконується на однопроцесорній машині. Нехай перший потік починає довгу операцію введення/виведення, тоді другий потік отримує процесорний час. Таким чином, замість того щоб простоювати, як це було б у разі однопоточного варіанту, наша програма продовжує виконуватися.

У принципі того ж ефекту можна досягти і без потоків, використовуючи можливості асинхронного або неблокуючого введення-виведення. Перший спосіб грунтується на тому, що при спробі читання або запису, яка повинна заблокувати процес, операція приймається до виконання, але блокування не відбувається. Замість цього можна продовжувати обчислення або здійснювати введення-виведення іншими каналами. Операційна система сповістить нас про закінчення операції введення-виведення посиланням відповідного сигналу.

Другий же спосіб грунтується на тому, що час від часу за допомогою спеціального системного виклику додаток опитує файл, в який необхідно здійснювати уведення-виведення, – чи готові вони до виконання тієї або іншої операції. У разі готовності додаток може виконувати бажану операцію. Ясно, що в проміжках між опитуваннями додаток може виконувати якусь імовірно корисну роботу.

Проте обидва ці способи мають істотний недолік. Код, що виходить при їх реалізації, досить складний, оскільки в ньому доводиться розносити введення-виведення і обробку даних.

Особливу гостроту проблема введення-виведення набуває в мережевих додатках, де блокування однопотокового процесу повільною операцією введення-виведення в мережі при обслуговуванні одного із запитів веде до

призупинення обслуговування інших запитів. Як правило, така поведінка для цих застосувань не прийнятна. Саме тому, найчастіше використовувалися можливості конкурентного виконання запитів, спочатку засновані на використанні багатьох процесів, потім на використанні багатьох потоків або неблокуючого введення-виведення.

Навіть якщо програма в багатопотоковому варіанті не працюватиме продуктивніше, то все одно можна отримати деякі переваги від використання потоків. Виділяючи в програмах незалежні події і послідовності подій, і оформляючи їх у вигляді різних потоків, можна отримати програми, які краще відображають реальність, і як наслідок, які буде зручніше супроводжувати.

Природно, що за все хороше доводиться платити. Чим же нам доводиться платити в разі використання потоків?

По-перше, можливою **втратою продуктивності**. Очевидно, що навіть повністю розпаралелюваний обчислювальний додаток на однопроцесорній машині, у багатопотоковому варіанті виконуватиметься повільніше за свою однопоточну версію. Це обумовлено накладними витратами на управління потоками і забезпечення узгодження між ними (на синхронізацію). Далі, навіть при виконанні на багатопроцесорній машині в разі коду, що погано розпаралелений, ми можемо виявитися в ситуації, коли згадані накладні витрати перевищать виграш від використання паралелізму.

По-друге, написання багатопотокових програм вимагає більшої акуратності, ніж написання звичайних програм. До стандартних проблем, таким як вихід за межі своєї пам'яті, висячі посилання тощо додаються різноманітні проблеми, які пов'язані з паралельним програмуванням такі як тупіки, умови перегонів (race conditions) і багато інших (**складність кодування)**.

По-третє, багатопотокові програми з цілого ряду причин **важче відлагоджувати**, чим однопоточні програми. Тут позначається і деяка недорозвиненість засобів відладки таких додатків, і те, що сам процес відладки, змінюючи часові характеристики виконання програми, може призвести до її поведінки, абсолютно відмінної від того, при якому проявлялася помилка.

І, нарешті, той простий факт, що якщо раніше, наприклад, зіпсувати пам'ять міг тільки сам процес, то зараз це може зробити будь-який з декількох процесів. У результаті найпотужнішим засобом відладки подібних програм, як і раніше, залишається мозок програміста.

Тільки уважно зважуючи усі перераховані і деякі інші чинники, слід приймати рішення про те, чи варто використовувати потоки в програмі, чи ні.

6.7 Потоки у Windows 2000

Одним з основних понять, які вводяться при розгляді механізму організації мультизадачного режиму виконання програм в 32-х розрядних операційних системах MS Windows, є потік (thread).

Для кожного додатка Win32 ОС створює окремий процес. Контекст процесу включає віртуальний адресний простір додатка і ряд інших системних ресурсів, які використовуються потоками. Один або декілька потоків можуть

бути організовані всередині одного процесу. Такі потоки спільно використовують пам'ять і інші ресурси, виділені процесу. Перемикання потоків, які належать різним процесам, є складнішою процедурою, ніж перемикання потоків, які належать одному процесу.

6.7.1 Багатопоточність

Кожен процес повинен містити, принаймні, один потік який називається першим потоком (primary thread). Цей потік формується при створенні процесу. Інші потоки можуть бути створені згодом з будь-якого існуючого потоку процесу за допомогою функцій API [28]. Процеси в ОС Windows 2000 (W2K) організовані так, щоб забезпечити підтримку різних операційних середовищ. Процеси в різних середовищах відрізняються рядом параметрів, включаючи такі:
- іменування процесів;
- підтримка потоків у процесах;
- спосіб представлення процесів;
- спосіб захисту ресурсів процесів;
- взаємозв'язок процесів один з одним.

Відповідно структури і сервіси процесів, що надаються ядром W2K, порівняно прості і мають загальне призначення. Процеси реалізовані як об'єкти.

При вході користувача в систему створюється ознака доступу, куди входить ідентифікатор безпеки користувача. Кожен процес, який створюється цим користувачем або запускається ним, має копію цієї ознаки. Вказана ознака використовується ОС, щоб підтвердити можливість доступу користувача до захищених об'єктів, або можливість виконання спеціальних функцій в системі і в захищених об'єктах.

З процесами пов'язаний і ряд блоків, в яких визначається віртуальний адресний простір, закріплений в даний момент за процесом. Процес не може безпосередньо змінювати ці структури, в цьому він залежить від менеджера віртуальної пам'яті, який надає сервіс для виділення пам'яті процесу.

До складу процесу також входить таблиця об'єктів, яка управляє відомими процесу об'єктами. Для кожного потоку, що входить в цей процес, є один дескриптор. Крім того, процес має доступ до файлових об'єктів і до розділів спільно використовуваної пам'яті.

Кожен процес W2K включає такі компоненти (рис. 6.7):
- один або декілька потоків;
- віртуальний адресний простір, відмінний від адресних просторів інших процесів, за винятком випадків явного розподілу пам'яті;
- один або більше сегментів коду, включаючи код DLL;
- один або більше сегментів даних, що містять глобальні змінні;
- рядки з інформацією про змінні оточення, такі як поточний шлях пошуку файлу тощо;
- пам'ять купи (динамічні структури даних) процесу;
- ресурси процесу (відкриті дескриптори, файли, інші купи).

Рисунок 6.7 – Структура процесу і його потоків

Атрибути процесу і потоку в Windows наведені в таблиці 6.2.

Таблиця 6.2 – Атрибути процесу і потоку в Windows

Процес	Потік
Ідентифікатор процесу – унікальне значення, що ідентифікує процес в ОС	Ідентифікатор потоку – унікальне значення, що ідентифікує потік, коли він викликає сервіс.
Дескриптор захисту – описує, хто створив процес, права доступу тощо.	Контекст потоку – набір значень регістрів, якими визначається стан виконуваного потоку.
Базовий пріоритет – базовий пріоритет процесу.	Динамічний пріоритет – пріоритет виконуваного потоку в даний момент. Базовий пріоритет – нижній пріоритет динамічного пріоритету потоку.
Спорідненість процесів за замовчанням – заданий за замовчанням набір процесів, де можливе виконання потоків.	Спорідненість процесів за потоком – множина процесів, де можливе виконання потоків.
Час виконання – сумарний час, витрачений на виконання усіх потоків у процесі.	Час виконання потоку – сукупний час, витрачений на виконання потоку в режимі користувача і режимі ядра.
Лічильник уведення/виведення – змінні, в які заносяться відомості про кількість і тип операцій введення/виведення, виконаних потоками процесу.	Статус сповіщення – це прапор, який вказує, чи слід потоку виконувати асинхронний виклик процедури.
Лічильник операцій з віртуальною пам'яттю – це змінні, в які заносяться відомості про кількість і тип операцій з віртуальною пам'яттю, виконаних потоками процесу.	Лічильник призупинень – в ньому вказується, скільки разів виконання потоку було призупинене без подальшого відновлення.

Продовження таблиці 6.2

Процес	Потік
Квоти – максимальна кількість сторінкової пам'яті і процесорного часу доступного процесу.	Маркери режиму анонімного втілення – це часова ознака доступу.
Порти виключення/відладки – це канали обміну інформацією між процесами, в які диспетчер процесів повинен відправляти повідомлення при виникненні виняткових ситуацій.	Порт завершення – це канал обміну інформацією між процесами, куди диспетчер процесів відправляє повідомлення при завершенні потоку.
Статус виходу – причини завершення процесу	Статус виходу – причини завершення потоку

Усі потоки процесу спільно використовують код, глобальні змінні і ресурси процесу. Кожен потік планується незалежно.

Деякі атрибути потоку подібні до атрибутів процесів. Значениия таких атрибутів потоку витягаються зі значень атрибутів процесу. Наприклад, у багатопроцесорній системі споріднені процесори за потоком – це декілька процесорів, на яких може виконуватися цей потік. Вони співпадають з множиною процесорів споріднених по процесу або є його підмножиною. Інформація, що міститься в контексті потоку, дозволяє ОС призупиняти і поновлювати потоки. Основні функції управління процесами і потоками показані в таблиці. 6.3.

Таблиця 6.3 – Сервіси процесу і потоку в Windows

Процес	Потік
Створення процесу **CreateProcess().**	Створення потоку **CreateThread().**
Відкриття процесу **OpenProcess().**	Відкриття потоку **OpenThread()**
Інформація за запитом процеса	Інформація за запитом потока
Інформація з наладки процеса	Інформація з наладки потока
Поточний процес **GetCurrentProcessID()**	Поточний потік **GetCurrentThreadID()**
Припинення процесу **ExitProcess()**	Завершення потоку **ExitThread()**
	Отримання контексту
	Установка контексту
	Призупинення потоку **Delay()**
	Відновлення потоку
	Сповіщення потоків
	Перевірка сповіщення потоку

ОС Windows 2000 підтримує паралельне виконання процесів, оскільки потоки різних процесів можуть виконуватися одночасно. Більш того, декільком потокам одного і того ж процесу можуть бути виділені різні процесори, і ці потоки також можуть виконуватися одночасно. Потоки одного і того ж процесу можуть обмінюватися між собою інформацією за допомогою загального адресного простору і мають доступ до загальних ресурсів процесу.

6.7.2 Стани потоків в ОС Windows

При запуску додатка Win32 операційна система автоматично створює новий процес і перший потік процесу. Усі інші потоки можуть бути створені з існуючих потоків. Другий потік може бути створений з першого, третій з першого або другого тощо. Немає ніяких обмежень того, з якої точки програми створюється новий потік.

В ОС W2K потік у ході свого існування може мати один з шести станів (рис. 6.8) [10]. Життєвий цикл потоку починається в той момент, коли програма створює новий потік. Менеджер процесів виділяє пам'ять для об'єкту-потоку і звертається до ядра, щоб ініціалізувати об'єкт-потік ядра.

Готовність. При пошуку потоку на виконання диспетчер переглядає тільки потоки, що знаходяться в стані готовності, в яких є все для виконання, але бракує тільки процесора.

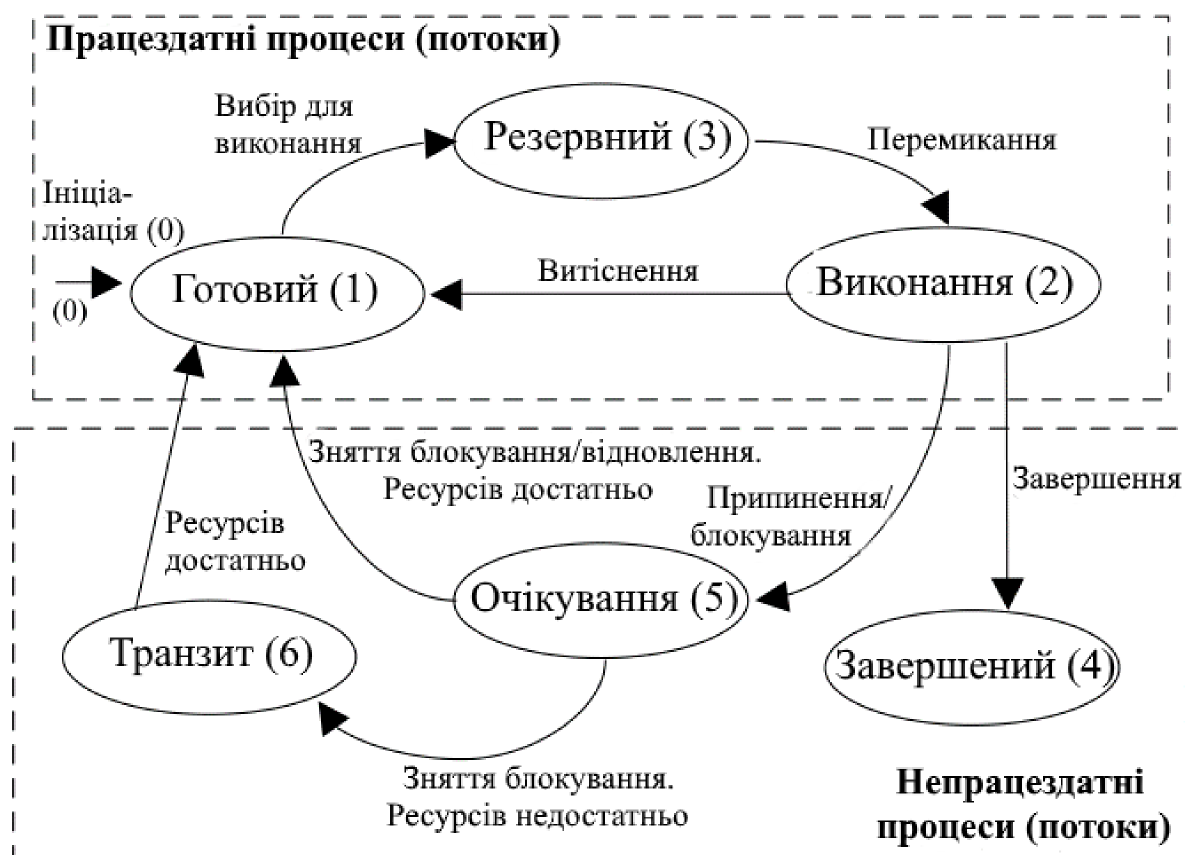


Рисунок 6.8 – Стан потоків в ОС Windows

Резервний (Першочергова готовність). Для кожного процесора системи вибирається один потік, який виконуватиметься наступним (найперший потік в черзі). Коли умови дозволяють, відбувається перемикання на контекст цього потоку.

Виконання. Як тільки відбувається перемикання контекстів, потік переходить у стан виконання і знаходиться в ньому до тих пір, поки або ядро не витіснить його через те, що з'явився пріоритетніший потік, або закінчився квант

часу, виділений цьому потоку, або потік завершиться взагалі, або він за власною ініціативою перейде в стан очікування.

Очікування. Потік може входити в стан очікування декількома способами. Потік за своєю ініціативою чекає деякий об'єкт для того, щоб синхронізувати своє виконання. Операційна система (наприклад, підсистема введення-виведення) може чекати в інтересах потоку. Підсистема оточення може безпосередньо змусити потік призупинити себе. Коли очікування потоку добіжить кінця, він повертається в стан готовності.

Транзит (Перехідний стан). Потік входить в перехідний стан, якщо він готовий до виконання, але ресурси, які йому потрібні, зайняті. Наприклад, сторінка, що містить стек потоку, може бути вивантажена з оперативної пам'яті на диск. При звільненні ресурсів потік переходить в стан готовності.

Завершення. Коли виконання потоку закінчилося, він входить в стан завершення. Знаходячись в цьому стані, потік може бути або видалений, або не видалений. Це залежить від алгоритму роботи менеджера об'єктів, відповідно до якого він і вирішує, коли видаляти об'єкт.

Виконання потоку завершується, коли функція потоку повертає управління. Це нормальний і рекомендований спосіб завершення роботи потоку. Проте в ряді випадків застосовуються інші методи. Потік може бути завершений функціями ExitThread, TerminateThread або функціями ExitProcess, TerminateProcess. Остання пара функцій закриває потік разом з процесом, якому він належить. Функції ExitXXX «правильніші». TerminateThread і TerminateProcess використовуються тільки в критичних обставинах, наприклад при завершенні потоків і процесів у разі помилки, після якої нормальне функціонування програми неможливе.

6.7.3 Підтримка симетричної багатопроцесорної обробки

Windows 2000 може працювати в обчислювальній системі з симетричною багатопроцесорною архітектурою. У такій системі потоки можуть паралельно виконуватися на декількох процесорах.

ОС Windows 2000 підтримує симетричну багатопроцесорну конфігурацію апаратного забезпечення. Потоки будь-якого процесу, включаючи потоки виконавчої системи, можуть виконуватися на будь-якому процесорі. За умови відсутності обмежень на засіб процесорів мікроядро виділяє готовому до виконання потоку процесор, який звільняється першим. При цьому гарантується, що жоден процесор не простоюватиме або не виконуватиме потік з нижчим пріоритетом, якщо готовий процес з вищим пріоритетом.

У Windows 2000 потік має додатковий атрибут маску процесорів (affinity). Маска визначає підмножину мікропроцесорів, на яких може виконуватися потік. Цей атрибут задається функцією SetThreadAffinity.

DWORD SetThreadAffinityMask (
HANDLE *hThread***,** // посилання на потік
DWORD *dwThreadAffinityMask* // маска affinity**);**

Маска може бути задана відразу для всіх потоків, які належать одному процесу. Для цього використовується функція SetProcessAffinityMask. Окрім маски для потоку можна програмно задати так званий ідеальний процесор (ideal processor). Ідеальний процесор – це процесор, який має перевагу для виконання потоку. Якщо в момент перемикання потоку вільні декілька процесорів, включаючи ідеальний, потік виконуватиметься на ідеальному процесорі. Ідеальний процесор призначається функцією SetThreadIdealProcessor.

DWORD SetThreadIdealProcessor(
HANDLE *hThread***,** // посилання на потік
DWORD *dwIdealProcessor***);** // номер ідеального процесора

Наведені вище операції допустимі тільки в Windows, починаючи з технології NT. Наприклад, ОС Windows 95 розрахована на роботу в системі з одним процесором.

6.8 Управління процесами і потоками в LINUX

6.8.1 Процеси в LINUX

В ОС LINUX процес, або задача, подаються як структура даних. LINUX підтримує таблицю задач, що є списком покажчиків на кожну визначену в даний момент структуру даних. У цій структурі даних інформація розбита на такі категорії.

Стан. Стан виконання процесу (виконується, готовий до виконання, призупинений, зупинений, зомбі);

Інформація з планування. Інформація, яка потрібна ОС LINUX для планування процесів. Процес може бути звичайним або таким, що виконується в реальному часі. Крім того, він має деякий пріоритет. Процеси, що виконуються в реальному часі, плануються до звичайних процесів. Лічильник веде відлік часу, відведеного процесу.

Ідентифікатори. Кожен процес має свій власний ідентифікатор, а також ідентифікатори користувача і групи. Ідентифікатор групи використовується для того, щоб призначити групі користувачів права доступу до ресурсів.

Обмін інформацією між процесами. В ОС LINUX використовується такий же механізм міжпроцесної взаємодії, як і в ОС UNIX SVR4.

Зв'язки. Кожен процес містить у собі зв'язки з паралельними до нього процесами, із спорідненими йому процесами (з якими він має загальний батьківський процес) і зв'язки з усіма своїми дочірніми процесами.

Час і таймери. Сюди входить час створення процесу, а також кількість процесорного часу, витраченого на цей процес. З процесом також можуть бути пов'язані інтервальні таймери (кванти часу, один або декілька). Квант часу задається в процесі за допомогою системного виклику. Після закінчення кванта часу процесу подається відповідний сигнал. Таймер може бути створений для одноразового або періодичного використання.

Файлова система. Містить у собі покажчики на всі файли, які відкриті цим процесом.

Віртуальна пам'ять. Визначає відведену цьому процесу віртуальну пам'ять.

Контекст, залежний від процесора. Інформація про регістри і стек, що становить контекст цього процесу.

Створення процесу. Процес породжується за допомогою системного виклику fork(). При цьому виклику відбувається перевірка на наявність вільної пам'яті, доступної для розміщення нового процесу. Якщо необхідна пам'ять доступна, то створюється процес-нащадок поточного процесу, що є точною копією цього процесу (клонований або народжений процес). При цьому в таблиці процесів для нового процесу будується відповідна структура. Нова структура створюється також в таблиці користувача. При цьому всі її змінні ініціалізуються нулями. Цьому процесу привласнюється новий унікальний ідентифікатор, а ідентифікатор батьківського процесу запам'ятовується в блоці управління процесом.

Завершення процесу. Для завершення процесу використовується системний виклик exit(), при якому звільняються усі використовувані ресурси, такі як пам'ять і структури таблиць ядра. Крім того, завершуються і процеси-нащадки, породжені цим процесом.

Потім з пам'яті видаляються сегменти коду і даних, а сам процес переходить в стан ***зомбі*** (у полі *Stat* такі процеси позначаються буквою «Z»). Зомбі не займає процесорного часу, але рядок в таблиці процесів залишається, і відповідні структури ядра не звільняються. Після завершення батьківського процесу зомбі, що «осиротів», на короткий час стає нащадком init, після чого вже «остаточно помирає». І, нарешті, батьківський процес повинен очистити всі ресурси, займані дочірніми процесами.

Якщо батьківський процес з якоїсь причини завершиться раніше дочірнього, останній стає «***сиротою»*** (orphaned process). Такі «сироти» також автоматично «усиновляються» програмою *init*, що виконується в процесі з номером 1, яка і приймає сигнал про їх завершення.

Також, процес може впасти в «сон», який не вдається перервати (у полі Stat це позначається буквою «D»). Процес, що знаходиться в такому стані, не реагує на системні запити і може бути знищений тільки перезавантаженням системи.

Взаємодія процесів. Найпоширенішим засобом взаємодії процесів є ***сокети*** *(sockets)*. Програми підключаються до сокета і видають запит на прив'язку до потрібної адреси. Потім дані передаються від одного сокета до іншого відповідно до вказаної адреси. Сигнал інформує інший процес про виникнення певних умов усередині поточного процесу, що вимагають реакції поточного процесу. Багато програм обробки сигналів для аналізу виниклої проблеми виводять дамп пам'яті.

Канали реалізовані в двох класах. Перший з них створюється за допомогою системного виклику *pipe()*. При цьому для обміну інформацією між процесами ініціалізувалася спеціальна структура в ядрі. Потім, коли процес породжує новий процес, між двома процесами відкривається комунікаційний канал. Іншим типом каналів є ***іменовані канали***. При їх використанні із структурою, що управляє, в ядрі зв'язується спеціальний каталог, через який два автономні процеси можуть

обмінюватися даними. При цьому, кожен процес повинен відкрити канал у вигляді звичайних файлів (один для читання, інший для запису). Потім операції введення-виведення виконуються звичайним способом.

Черга повідомлень є механізмом, коли один процес надає блок даних зі встановленими прапорами, а інший процес розшукує блок даних, прапори якого встановлені в необхідних значеннях.

Семафори є засобом передачі прапорів від одного процесу до іншого. «Піднявши» семафор, процес може повідомити, що він знаходиться в певному стані. Будь-який інший процес в системі може відшукати цей прапор і виконати необхідні дії.

Спільно використовувана пам'ять дозволяє процесам отримати доступ до однієї і тієї ж області фізичної пам'яті.

6.8.2 Потоки в LINUX

Поняття процесу і потоку в LINUX дуже тісно пов'язані і тому їх важко відрізнити, потоки навіть часто називають легковаговими процесами.

Основні відмінності процесу від потоку полягають у тому, що кожному процесу відповідає своя незалежна від інших область пам'яті, таблиця відкритих файлів, поточна директорія і інша інформація рівня ядра. Потоки ж не пов'язані безпосередньо з цими сутностями. В усіх потоків, тих, що належать цьому процесу, все вище перераховане загальне, оскільки належить цьому процесу.

Для управління потоками використовуються відповідні засоби, які доступні програмістові через мовні конструкції, системні виклики ОС або спеціально розроблені бібліотеки. Наприклад, бібліотека потоків **Pthread** визначає об'єкт атрибутів потоку, що інкапсулює властивості потоку, до яких творець об'єкту може отримати доступ і модифікувати їх.

Створення потоку і ідеологія POSIX API. При вибраному для вивчення низькорівневому підході до підтримки потоків у мові усі операції, пов'язані з ними, виражаються явно через виклики функцій мови *C* і інтерфейси підтримки потоків відповідному стандарту **POSIX API**. Згідно з ним потік (нитка) створюється за допомогою такого виклику:

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void* (*start)(void *), void *arg)

Спрощено виклик **pthread_create(&thr, NULL, start, NULL)** створить потік який почне виконувати функцію *start* і запише в змінну **thr** ідентифікатор створеного потоку. На прикладі цього виклику детально розглянемо декілька допоміжних концепцій POSIX API з тим, щоб не зупинятися на них далі.

Перший аргумент цієї функції **thread** – це покажчик на змінну типу **pthread_t**, в яку буде записаний ідентифікатор створеного потоку, який потім можна буде передавати іншим викликам, коли неоюхідно зробити що-небудь з цим потоком.

Другий аргумент цієї функції **attr** – це покажчик на змінну типу **pthread_attr_t**, яка задає набір деяких властивостей створюваного потоку.

Третій аргумент виклику **pthread_create** – це покажчик на функцію типу **void*().** Саме цю функцію і починає виконувати знову створений потік. При цьому, в якості параметра цієї функції передається четвертий аргумент виклику **pthread_create**.

Функція **pthread_create** повертає нульове значення в разі успіху і ненульовий код помилки в разі невдачі. Це також одна з особливостей POSIX API. Замість стандартного для Unix підходу, коли функція повертає лише деякий індикатор помилки, а код помилки встановлює в змінній **errno**, функції Pthreads API повертають код помилки в результаті свого аргументу.

Життєвий цикл потоку. Розглянемо тепер життєвий цикл потоку, а саме послідовність станів, в яких знаходиться потік за час свого існування.

Потік може перебувати в одному з чотирьох станів: готовності, виконання, останову-очікування, блокування (рис.6.9).

Готовий. Потік знаходиться в стані готовності, коли він готовий до виконання. Можливо, він тільки що був створений, або був витіснений з процесора іншим потоком, або тільки що був розблокований (вийшов з відповідного стану).

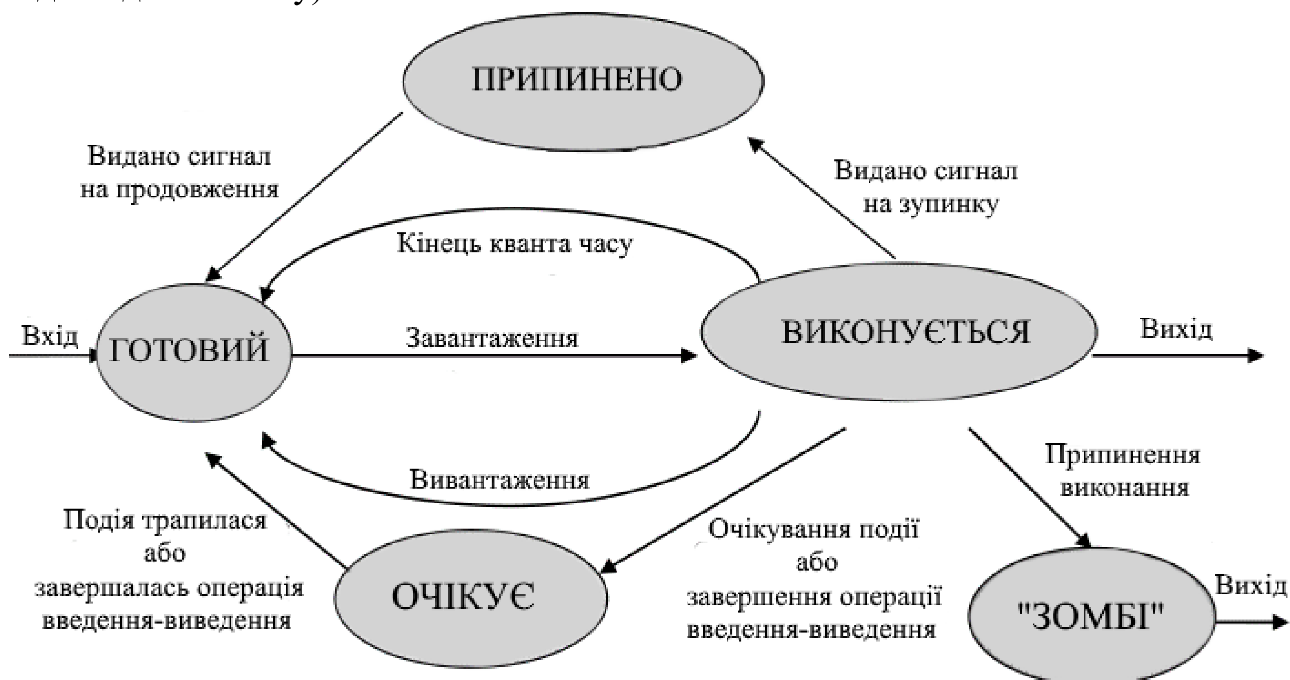


Рисунок 6.9 – Стани потоків в средовищах Unix/Linux

Усі готові до роботи потоки поміщаються в черги готовності згідно зі своїми пріоритетами. Потік переходить в стан *виконання*, коли він вибирається з черги і назначається процесору. Потік знімається з процесора і поміщається в чергу готових потоків, якщо його квант часу закінчився або якщо він перейшов в стан готовності з великим пріоритетом. Потік готовий до виконання, але чекає процесора.

Виконується. Потік, що виконується, може отримати сигнал і перейти в стан останову, який принципово відрізняється від стану очікування. Потік отримує сигнал зупинитися, якщо він знаходиться в стані відладки або із-за виникнення особливої ситуації в системі. Пізніше потік може бути розбуджений

або ліквідований. Якщо потік не є відкріпленим, то після завершення виконання він переходить в спеціальний стан *«зомбі»*. В цьому стані він вже не здатний продовжувати виконання, він також не використовує системних ресурсів, але не може покинути систему, поки його батьківський потік не поміняє його статус на завершення.

Чекає. Потік, що чекає, може бути в двох підстанах:

1. *Такий, що переривається*. Це стан блокування, в якому процес чекає на подію (наприклад, завершення операції введення-виведення, звільнення ресурсу або сигналу від іншого процесу).
2. *Такий, що не переривається*. Це стан блокування іншого роду. Його відмінність від попереднього полягає в тому, що в цьому стані процес безпосередньо чекає виконання якоїсь апаратної умови, тому він не сприймає ніяких сигналів.

Зупинений. Процес (потік) був зупинений і може бути продовжений тільки при відповідній дії іншого процесу. Наприклад, процес знаходиться в стані відладки, може перейти в стан зупинки.

Завершення потоку, особливості головного потоку. Потік завершується, коли відбувається повернення з функції **start**. При цьому якщо ми хочемо отримати повернене значення функції, то ми повинні скористатися функцією:

int pthread_join(pthread_t thread, void value_ptr)**

Ця функція чекає завершення потоку з ідентифікатором **thread**, і записує її повернене значення в змінну, на яку вказує **value_ptr.** При цьому, звільняються всі ресурси, пов'язані з потоком, і, отже, ця функція може бути викликана для цього потоку тільки один раз.

Окрім повернення з функції потоку, існує ще один спосіб завершити потік, а саме – викликати **exit(),** аналогічний викличу **exit()** для процесів:

int pthread_exit(void *value_ptr)

Цей виклик завершує виконуваний потік, повертаючи в якості результату його виконання **value_ptr.** Реально, що при виклику цієї функції, потік з неї просто не повертається. Потрібно звернути також увагу на той факт, що функція **exit()** як і раніше завершує процес, тобто, у тому числі, знищує всі потоки.

Контрольні питання і тести до розділу 6

Контрольні питання

1. Для чого в операційних системах почали застосовувати нову одиницю роботи – потоки?
2. Назвіть синоніми для терміну «потоки».
3. Як виконуються потоки багатопотокового процесу в системі з одним процесором?
4. Які ресурси процесу можуть використовувти потоки?
5. Перелічить елементи індивідуальні для кожного потоку.
6. Які труднощі можуть виникнути при використанні потоків для розв'язання проблем паралелізму?
7. Чи можуть потоки, як і процеси, породжувати потоки-нащадки?

8. Перелічить основні стани потоку.
9. Які існують загальні категорії потоків?
10. За допомогою яких засобів можна створити додаток, щоб він був багатопотоковим на рівні користувача?
11. Чи створює ОС таблицю потоків для потоків, працюючих в просторі користувача?
12. Яка інформація надається ядру ОС про потоки, працюючі на рівні користувача?
13. Назвіть переваги використання потоків на рівні користувача перед використанням потоків на рівні ядра:
14. Які два явні недоліки властиві потокам на рівні користувача в порівнянні з використанням потоків на рівні ядра?
15. Від яких недоліків потоків на рівні користувача можна позбавитися при використанні потоків на рівні ядра?
16. У яких системах, найчастіше, використовуються спливаючі потоки?
17. Що розуміється під асинхронними подіями?
18. У чому складність створення переносимих багатопотокових додатків, працюючих на рівні користувача?
19. Чому взаємодія потоків одного процесу між собою ефективніше, ніж взаємодія окремих процесів?
20. Чому потоки рівня користувача мають кращу переносимість в порівнянні з потоками рівня ядра?

Тести

1. Які компоненти є унікальними для кожного потоку (нитки, threads) виконання одного процесу?
   1) локальні і глобальні змінні і регістри;
   2) системні ресурси і вміст регістрів;
   3) глобальні змінні і стек;
   4) програмний лічильник, стек і вміст регістрів.
2. Коли процес (потік), що знаходиться в стані «закінчив виконання», може остаточно покинути систему?
   1) після певного інтервалу часу;
   2) тільки при перезавантаженні операційної системи;
   3) після завершення процесу-батька;
   4) після блокування процесу-батька.
3. Який стан не визначений для потоку в системі?
   1) очікування;
   2) синхронізація;
   3) виконання;
   4) готовність.
4. Потік переходить із стану виконання в стан очікування в результаті:
   1) виникнення помилки;
   2) очікування завершення введення-виведення або іншої події;
   3) після закінчення кванта часу;

4) витіснення іншим потоком.

5. Потік, який вичерпав свій квант часу, переводиться в стан:
   1) очікування;
   2) готовності;
   3) завершення.
6. Деяка сутність усередині процесу, яка одержала процесорний час називається:
   1) задача;
   2) процес;
   3) завдання;
   4) потік.
7. При створенні потоку ОС відразу переводить його в стан:
   1) очікування;
   2) виконання;
   3) готовності.
8. В ОС, що підтримують потоки виконання (threads) усередині одного процесу на рівні ядра системи, процес знаходиться в стані «готовність», якщо:
   1) хоч би один потік знаходиться в стані готовність, і немає жодного потоку в стані очікування;
   2) хоч би один потік процесу знаходиться в стані готовність;
   3) хоч би один потік процесу знаходиться в стані готовність, і немає жодного потоку в стані виконання.
9. Потік, що зробив синхронний (блокуючий) виклик, переводиться планувальником ОС в стан очікування, а після завершення обробки виклику – в стан:
   1) очікування;
   2) синхронізація;
   3) виконання;
   4) готовність.
10. Створення ... вимагає від ОС менших накладних витрат, чим створення процесу.
    1) задачі;
    2) процесу;
    3) завдання;
    4) потоку.
11. Укажіть три причини на користь потоків, для яких слід застосовувати потоки замість процесів:

1. Для потоків є спеціальні механізми обміну повідомленнями і даними;

2. Внаслідок того, що потоки, які відносяться до одного процесу, виконуються в одному віртуальному адресному просторі, між ними легко організувати тісну взаємодію;

3. Легкість створення і знищення потоків, оскільки з потоком не пов'язані ніякі ресурси;

4. Використання декількох потоків може підвищити продуктивність програми, що виконує складні розрахунки в однопроцесорних системах;

5. Використання потоків на рівні користувача може скоротити кількість переривань (режим ядра – режим користувача).

1) 1, 3,5;
2) 2,3,5;
3) 1, 2, 5;
4) 3, 4, 5.

12. Нижче пераховані елементи, що спільно використовуються всіма потоками процесу, і елементи індивідуальні для кожного потоку. Вкажіть типові елементи, індивідуальні для кожного потоку.
    1) інформація про використання ресурсів, Лічильник команд, Регістри, Стек;
    2) Лічильник команд, Адресний простір, Стек, Глобальні змінні;
    3) Лічильник команд, Регістри, Стек, Стан;
    4) Стан, Лічильник команд, Регістри, Відкриті файли.
13. Перелічить три основні переваги потоків на рівні користувача в порівнянні з потоками на рівні ядра:

    1. Перемикання потоків не включає перехід в режим ядра.

    2. У стратегії з наявністю потоків тільки на рівні користувача процес може скористатися перевагами багатопроцесорної системи.

    3. Планування здійснюється залежно від специфіки додатку.

    4. Використання потоків на рівні користувача застосовне для будь-якої ОС.

    5. Коли в потоці, який працює в режимі користувача, виконується системний виклик, то блокується тільки цей потік.
    1) 1, 3, 4;
    2) 2, 3,4;
    3) 1, 3, 5;
    4) 2, 3,4.
14. Перелічить два основні недоліки потоків на рівні користувача в порівнянні з потоками на рівні ядра:

    1. Перемикання потоків не включає перехід в режим ядра.

    2. У стратегії з наявністю потоків тільки на рівні користувача процес не може скористатися перевагами багатопроцесорної системи.

    3. Планування здійснюється залежно від специфіки додатка.

    4. Коли в потоці, який працює в режимі користувача, виконується системний виклик, то блокується не лише цей потік, але і усі потоки того процесу, до якого він належить.

    5. Використання потоків на рівні користувача застосовне для будь-якої ОС.
    1) 1, 2;
    2) 2, 4;
    3) 4, 5;
    4) 2, 3.
15. При порівнянні потоків на рівні користувача і потоків на рівні ядра згадувалося, що недолік потоків на рівні користувача полягає в тому, що

виконання системного виклику блокує не лише цей потік, але і усі інші потоки цього процесу. Чому так відбувається? Тому що:

1) цей потік забув передати сигнал іншим потокам (програміст забув вставити відповідну команду), що він робить системний виклик;
2) в цей час інші потоки цього процесу не могли заволодіти головним рессурсом – процесором;
3) інші потоки повинні дочекатися тієї інформації, яку зажадав цей потік своїм системним викликом, щоб потім нею скористатися;
4) ОС нічого не знає про потоки на рівні користувача, для неї відомий тільки один головний потік (процес), системний виклик якого вона і обробляє;
5) ОС нічого не знає про усі потоки на рівні користувача, але системний виклик зробив головний потік (процес), тому потрібно заблокувати усі потоки.

16. Якщо процес завершується, але якісь його потоки, які виконуються на рівні ядра, все ще виконуються, то чи будуть вони виконуватися і далі?
    1) ні, процес знаходиться в стані «закінчив виконання» (тобто далі виконуватися не буде), якщо процес завершується;
    2) так, процес знаходиться в стані «закінчив виконання», якщо усі його потоки знаходяться в стані «закінчив виконання»;
    3) ні, процес знаходиться в стані «закінчив виконання» (тобто далі виконуватися не буде), оскільки він є головним потоком для ОС;
    4) так, процес завершується, але він продовжує виконуватися далі, оскільки повинен дочекатися інформації від потоку, що виконується в даний момент.
17. Виберіть вірне висловлювання:
    1) потік може містити декілька процесів;
    2) потік і процес є рівноправними одиницями роботи;
    3) процес є різновидом потоку;
    4) процес може містити декілька потоків.
18. Чи вірно, що взаємодія процесів ефективніша, ніж взаємодія потоків одного процесу?
    1) так;
    2) ні.
19. Чи можуть багатопотокові додатки виконуватися швидше за однопотокових?
    1) так;
    2) ні.

7 ВЗАЄМНІ ВИКЛЮЧЕННЯ І БАГАТОЗАДАЧНІСТЬ

Основні питання, на які зосереджується увага розробників ОС, пов'язана з управлінням процесами і потоками.

1. **Багатозадачність**: управління певною кількістю процесів в однопроцесорній системі.
2. **Багатопроцесорність**: управління певною кількістю процесів у багатопроцесорній системі.
3. **Розподілені обчислення:** управління певною кількістю процесів, що виконуються в розподіленій обчислювальній системі з множиною комп'ютерів (**кластери**).

Розглянемо в контексті багатопроцесорності і багатозадачності проблему паралельності виконання процесів. Основною вимогою підтримки паралельних процесів є можливість забезпечення взаємовиключення, тобто можливість забезпечити роботу тільки одного процесу з призупиненням виконання інших.

7.1 Принципи паралельних обчислень

У однопроцесорній багатозадачній системі процеси виконуються по черзі для створення ілюзії одночасного виконання. Попри те, що при цьому не досягається реальна паралельна робота процесів і, більше того, є певні накладні витрати, пов'язані з перемиканням між процесами, таке виконання, що чергується, забезпечує чималі вигоди з точки зору ефективності і структуризації програм.

У багатопроцесорних системах можливо не лише чергування процесів, але їх перекриття. Перекриття і чергування процесів є принципово різними режимами роботи, але їх можна розглядати як приклади паралельних обчислень, які породжують однакові проблеми.

7.1.1 Участь операційної системи

За наявності паралельних обчислень перелічимо такі питання, які виникають при створенні та управлінні ОС.

1. ОС повинна відстежувати різні активні процеси. Це виконується за допомогою блоків управління процесами.
2. ОС повинна розподіляти і звільняти різні ресурси для кожного активного процесу, а саме:
   - процесорний час, це функція планування;
   - пам'ять, більшість ОС використовують схему віртуальної пам'яті;
   - файли системи;
   - пристрої введення-виведення.
3. ОС повинна захищати дані і ресурси кожного процесу від ненавмисного впливу інших процесів.
4. Результат роботи процесу не повинен залежати від швидкості його виконання стосовно інших процесів, які виконуються паралельно.

7.1.2 Взаємодія процесів

При необхідності використати один і той же ресурс паралельні процеси вступають в конфлікт один з одним. Кожному процесу по можливості надаються свої додаткові ресурси. Проте, для вирішення деяких задач процеси можуть об'єднувати свої зусилля. Зокрема, якщо два процеси бажають отримати доступ до одного ресурсу, то ОС виділить цей ресурс одному з процесів, тоді як другий процес змушений чекати на завершення роботи з ресурсом першого. Таким чином, швидкість виконання процесу, якому відмовлено в негайному доступі до ресурсу, зменшується.

Представимо таку ситуацію, що при банківській організації системи для обслуговування клієнтів виділяють окремий потік для кожного клієнта. Припустимо, що додавання даних на вклад клієнта зводиться до збільшення глобальної змінної Amount (Сума).

Розглянемо випадок, коли два клієнти *A* і *B* спільно користуються одним і тим же рахунком. Нехай на рахунку було 100 грн. Клієнт *A* хоче додати до рахунку 3 грн, а клієнт *B* – 5 грн. Потік **A** відповідає клієнтові *A*, а **B** – клієнтові *B*.

Спочатку розглянемо ситуацію, коли обчислювальна система має один ЦП. Передбачається, що ОС використовує витіснячий алгоритм планування. У цьому випадку основа проблеми полягає в можливості передачі ЦП від одного потоку до іншого до завершення їм операції із записом у змінну Amount. На рис. 7.1 представлені три можливих варіанти діаграми виконання потоків.

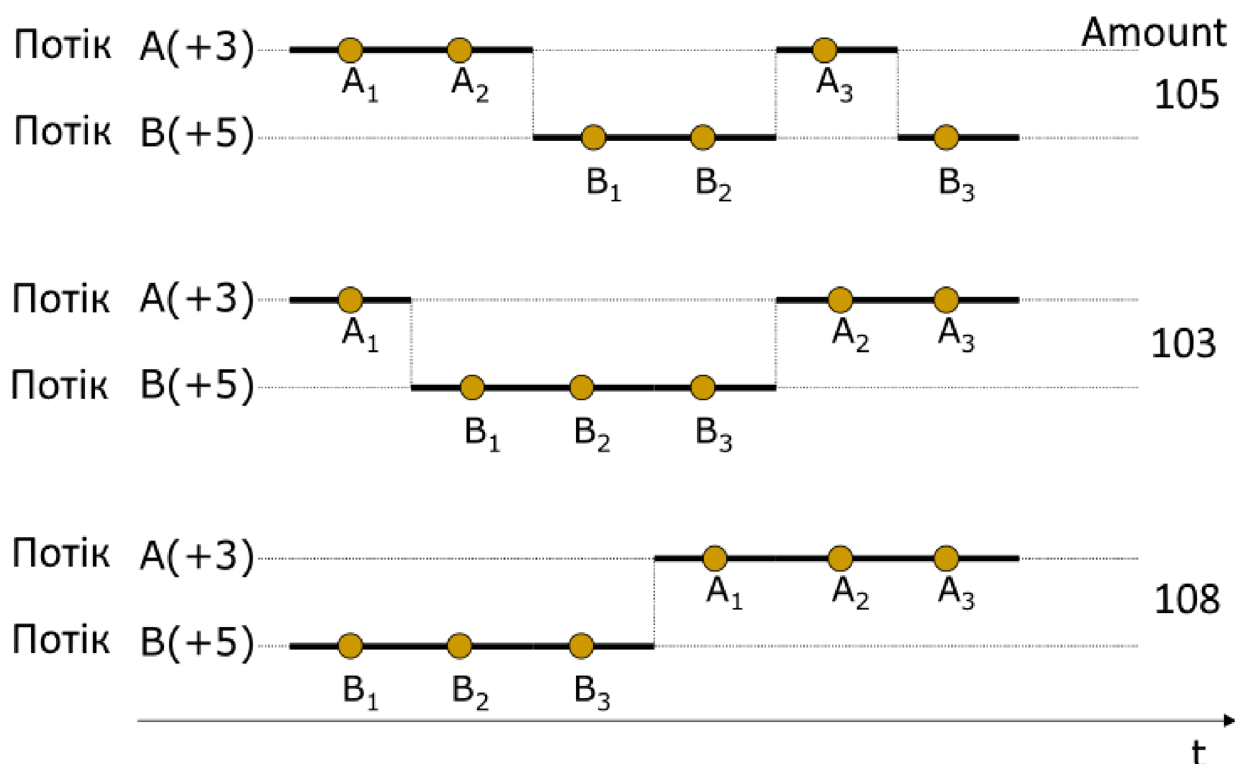


Рисунок 7.1 – Варіанти діаграми виконання потоків на однопроцесорній системі

Який саме варіант реалізується при конкретному запуску програми, залежить від взаємних швидкостей потоків і моментів передачі ЦП від одного потоку до іншого. Відзначимо, що в більшості випадків потік не може вплинути на дані фактори.

У разі багатопроцесорної системи також можливе виконання різних діаграм потоків (рис. 7.2).

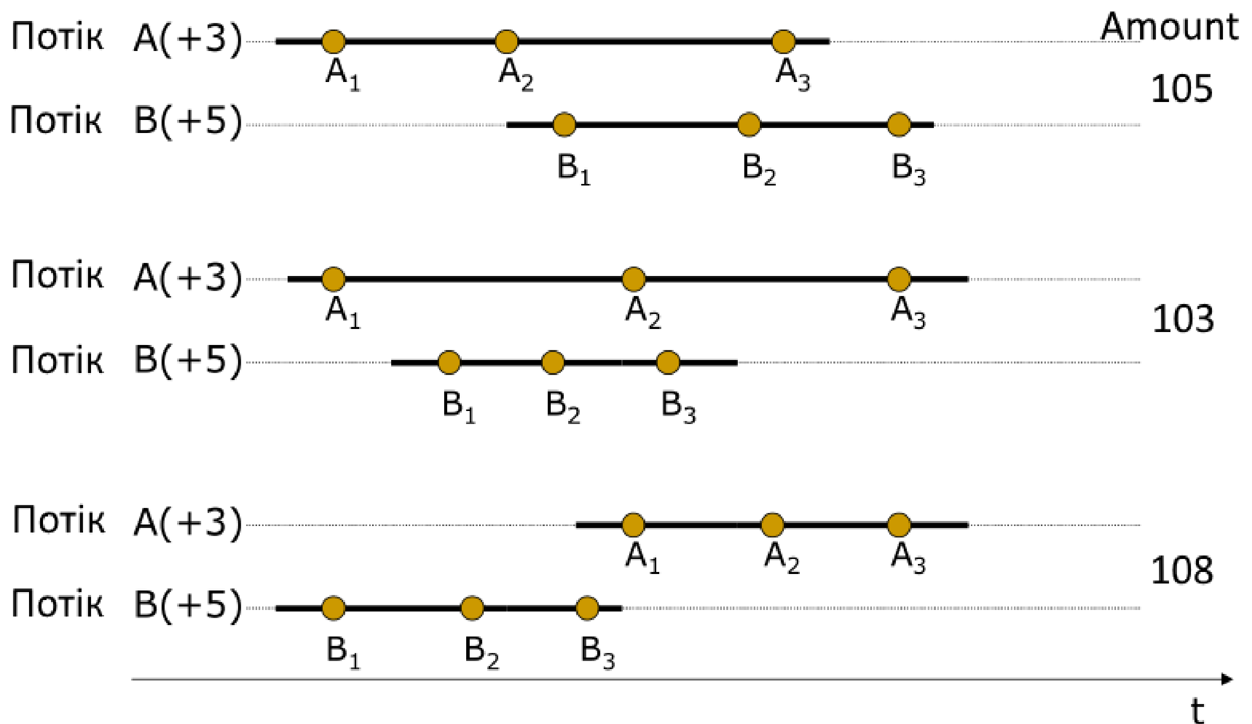


Рисунок 7.2 – Варіанти діаграми виконання потоків на багатопроцесорній системі

На даній діаграмі ми припускаємо, що потоки не витіснялися планувальником. У цьому випадку результат виконання операції залежить від відносних швидкостей потоків. Потік ніяк не може передбачити або вплинути на свою відносну швидкість: вона залежить від стану обчислювальної системи (наприклад, знаходяться його код або дані в кеш ЦП чи ні) і ОС та процесу-власника потоку (наприклад, частина сторінок, використовуваних потоком, може знаходитися на жорсткому диску).

Таким чином, результат обчислень у розглянутому прикладі не однозначний – все залежить від умов виконання потоків, і різні запуски програми можуть привести до різних результатів.

Такі ситуації, в яких два (і більше) потоків (процесів) прочитують або записують дані одночасно, і кінцевий результат залежить від того, який з них був першим, тобто від співвідношення швидкостей потоків, називають ***станом перегонів***, або ***змагань*** (race condition). Спроби рішення подібних проблем викликали необхідність **синхронізації процесів**. Тобто, основне призначення синхронізації – це необхідність забезпечення захисту даних від впливу інших потоків.

У разі конкуренції процесів виникають три проблеми.

Перша – необхідність ***взаємних виключень***. Припустимо, що два або декілька процесів вимагають доступу до одного ресурсу (введення-виведення). При виконанні кожен процес посилає команди в пристрій введення-виведення, отримує інформацію про його стан, посилає і/або отримує дані. Будемо говорити про такий ресурс як про ***критичний ресурс***, а про частину програми, яка його використовує, – як про ***критичний розділ*** (***критична секція***) програми.

Критичною секцією називається послідовність операторів, які мають доступ до об'єкта, що розділяється. Тобто, критична секція – це фрагмент коду потоку, який працює з ресурсом, що розділяється між декількома потоками. Тому говорять про зайняття і звільнення одним потоком критичної секції, загальної для декількох потоків (рис. 7.3) [9].

Украй важливо, щоб в критичній секції, пов'язаній з цим ресурсом, у будь-який момент часу міг знаходитися тільки один процес. Цей прийом називають ***взаємним виключенням*** або ***блокуванням***.

У разі блокування перевіряють, чи не було воно вже зроблено іншими процесами (потоками), а якщо це так, то цей процес переходить в очікування, інакше він здійснює блокування стану і входить в критичну секцію. Після виходу з критичної секції процес знімає блокування. Так реалізується взаємне виключення, звідси походить ще одна назва блокування – ***м'ютекс*** (mutex, скорочення від **mut**ual **ex**clusion).

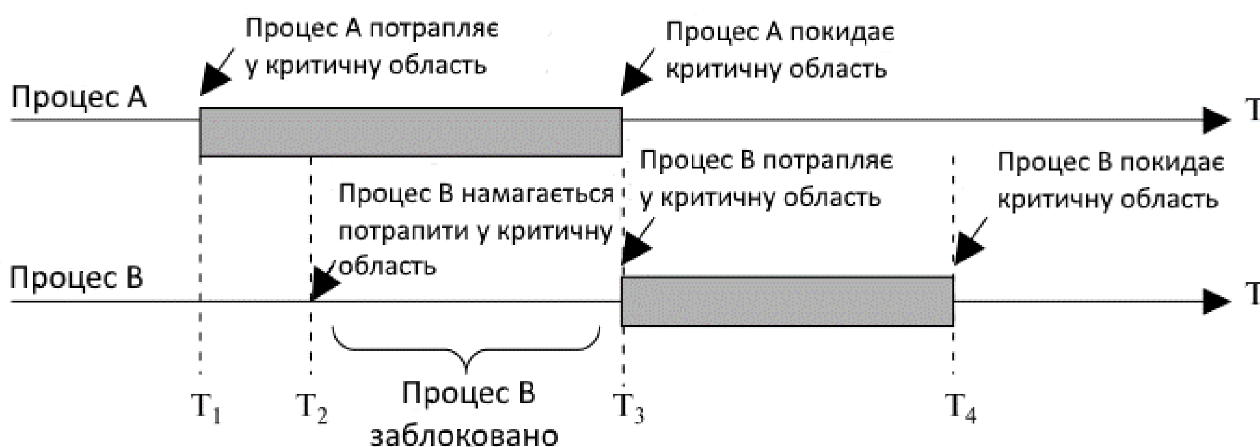


Рисунок 7.3– Критична секція

У загальному випадку м'ютексом називають примітив синхронізації, який не допускає виконання деякого фрагмента коду більше як одним процесом. М'ютекс – це змінна, яка може знаходитися в одному з двох станів: блокованому (0) або неблокованому ($\neq 0$). М'ютекс можна розглядати як спрощений семафор, який ми розглядатимемо пізніше. Звідси витікає, що вирішення проблеми змагань (race condition) є перетворенням коду в ***атомарну операцію***.

При реалізації взаємних виключень виникають додатково дві проблеми. Одна з них – ***взаємне блокування***. Розглянемо два процеси P1 і P2 і два ресурси R1 і R2. Припустимо, що кожному процесу для виконання деяких функцій потрібний доступ до обох ресурсів. При цьому можлива ситуація: ОС виділяє ресурс R1 процесу P2, а ресурс R2 – процесу P1. Процеси будуть взаємно заблоковані.

Остання проблема – ***голодування***. Припустимо, що ми маємо три процеси P1, P2, P3, кожному з яких потрібний доступ до ресурсу R. Припустимо, що P1 утримує ресурс R, а P2 і P3 призупинені в очікуванні звільнення ресурсу. Після виходу P1 з критичного розділу доступ до ресурсу отримує P2 або P3. Припустимо, що ОС надала доступ до ресурсу R процесу P3. Поки він працює з ресурсом, доступ до ресурсу знову потрібен процесу P1. У результаті може статися, що ОС знову надасть ресурс R процесу P1. Потім доступ знову може знадобитися P3. Таким чином, можлива ситуація, коли процес P2 ніколи не отримає ресурс R.

Управління конкуренцією неминуче призводить до участі ОС у цьому процесі, оскільки саме вона розподіляє ресурси. Процесам також потрібна можливість запитувати взаємовиключення, таке, як блокування ресурсу перед його використанням. Будь-яке розв'язання цього питання вимагає підтримки ОС, наприклад, такої, як забезпечення блокування.

Для усунення таких ситуацій може бути використаний так званий апарат подій. За допомогою цього засобу можуть вирішуватися не лише проблеми взаємного виключення, але й загальніші задачі синхронізації процесів. У різних ОС апарат подій реалізується по-своєму, але в будь-якому випадку використовуються системні функції аналогічного призначення.

7.1.3 Вимоги до взаємних виключень

Будь-яка можливість забезпечення підтримки взаємних виключень повинна відповідати таким вимогам, наведеним нижче.

1. Взаємовиключення повинно здійснюватися в примусовому порядку. У будь-який момент часу з усіх процесів, що мають критичний розділ для одного і того ж ресурсу, в цьому розділі може знаходитися тільки один процес.
2. Процес, що завершує роботу в некритичному розділі, не повинен впливати на інші процеси.
3. Не повинна виникати ситуація нескінченного очікування доступу до критичного розділу, тобто не повинні з'являтися взаємоблокування і голодування.
4. Коли в критичному розділі немає жодного процесу, будь-який процес, що запросив можливість входу в нього, повинен його негайно отримати.
5. Не повинно існувати ніяких припущень про відносні швидкості процесів, що виконуються, або число процесорів, на яких вони виконуються.
6. Процес залишається в критичному розділі впродовж обмеженого часу.

Є ряд способів задоволення перерахованих умов. Одним з них є передача відповідальності за відповідність вимогам самому процесу, який повинен виконуватися паралельно. Таким чином, процес, незалежно від того, чи є він системною програмою чи додатком, повинен координувати свої дії з іншими процесами для роботи взаємовиключень без підтримки з боку ОС.

7.2 Взаємне виключення через загальні змінні

Для синхронізації потоків одного процесу прикладний програміст може використати глобальні блокуючі змінні. З цими змінними, до яких усі потоки процесу мають прямий доступ, програміст працює не звертаючись до системних викликів ОС. Кожному набору критичних даних ставиться у відповідність двійкова змінна, якій потік привласнює значення 0, коли він входить в критичну секцію, і значення 1, коли він її покидає. На рис. 7.4 показаний фрагмент алгоритму потоку, що використовує для реалізації взаємного виключення доступу до критичних даних D блокуючу змінну F(D).

Рисунок 7.4 – Реалізація критичних секцій з використанням блокуючих змінних

Перед входом в критичну секцію потік перевіряє, чи не працює вже який-небудь потік з даними D. Якщо змінна F(D) встановлена в 0, то дані зайняті, і перевірка циклічно повторюється. Якщо ж дані вільні (F = 1), то значення змінної F(D) встановлюється в 0 і потік входить в критичну секцію. Після того як потік виконає усі дії з даними D, значення змінної F(D) знову встановлюється рівним 1. Наведемо приклад лістингу програми, що використовує загальні змінні для взаємного виключення (лістинг 7.1) [13]. Введемо булеву змінну mutEx, яка повинна набувати значення true(1), якщо входження в критичну секцію заборонене, або false(0), якщо входження дозволене.

Лістинг 7.1. Реалізація критичних секцій з використанням блокуючих змінних

```
1  static char mutEx = 0;
2  void csBegin ( void ) {
3     while ( mutEx );
4     mutEx = 1;
5  }
6  void csEnd ( void ) {
7     mutEx = 0;
8 }
```

При входженні у функцію csBegin процес потрапляє в цикл очікування (рядок 3), в якому знаходиться до тих пір, поки стан змінної виключення не дозволить йому увійти до критичної секції. Вийшовши з цього циклу, процес встановлює цю змінну в 1, забороняючи тим самим іншим процесам входити в їх критичні секції. Процес, який виконувався в критичній секції, при виході з останньої скидає змінну виключення в 0, дозволяючи цим іншим процесам входити в їх критичні секції.

Це рішення базується на неперервності доступу до пам'яті – до змінної mutEx, але воно є неправильним. Розглянемо такий випадок. Нехай процес *A* увійшов до своєї критичної секції і встановив mutEx=1. Поки процес *A* виконується всередині своєї критичної секції, два інші процеси – *B* і *C* також підійшли до своїх критичних секцій і звернулися до функції csBegin. Оскільки змінна mutEx встановлена в 1, процеси *B* і *C* зациклюються в рядку 3 коду функції csBegin. Коли процес *A* вийде зі своєї критичної секції і встановить mutEx=0, інший процес, наприклад *B*, вийде з циклу рядка 3. Але є ймовірність того, що перш, ніж процес *B* встигне виконати рядок 4 коду і цим заборонити вхід в критичну секцію іншим процесам, вийде з циклу рядка 3 і процес *C*. Таким чином, два процеси – *B* і *C* входять в критичну секцію, задача взаємного виключення не виконується.

Цей алгоритм можна удосконалити. Введемо для кожного процесу свою змінну, що відображає його знаходження в критичній секції. Ці змінні зведені в масив inside. Елемент масиву inside[i] має значення 1, якщо i-й процес знаходиться в критичній секції, і 0 – інакше (лістинг 7.2).

Лістинг 7.2. Покращений алгоритм реалізації критичних секцій з використанням блокуючих змінних

```
1  static char inside[2] = { 0,0 };
2  void csBegin ( int proc ) {
3    int competitor;
4    competitor = other ( proc );
5    do {
6      inside[proc] = 1;
7      if ( inside[competitor] ) inside[proc] = 0;
8     } while ( ! inside[proc] );
9  }
10 void csEnd (int proc ) {
11    inside[proc] = 0;
12 }
```

Процес встановлює свою ознаку входження (рядок 6). Але якщо він виявляє, що ознака входження конкурента теж встановлена (рядок 7), то він свою ознаку скидає. Ці дії повторюватимуться до тих пір, поки наш процес не збереже свою ознаку зведеного (рядок 8), а це можливо тільки в тому випадку, якщо ознака конкурента скинута. Це рішення не може бути прийняте ось з якої причини. Можливе таке співвідношення швидкостей процесів, при якому вони одночасно виконуватимуть рядок 7 – і одночасно скидати свої ознаки. Така «надмірна поступливість» процесів призведе до нескінченного відкладання рішення про вхід у критичну секцію.

7.3 Взаємні виключення з використанням системних функцій

Для усунення недоліків, які були описані в попередніх алгоритмах, у багатьох ОС передбачаються спеціальні системні виклики для роботи з критичними секціями. На рис. 7.5 показано, як за допомогою таких функцій реалізовано взаємне виключення в операційній системі Windows 2000.

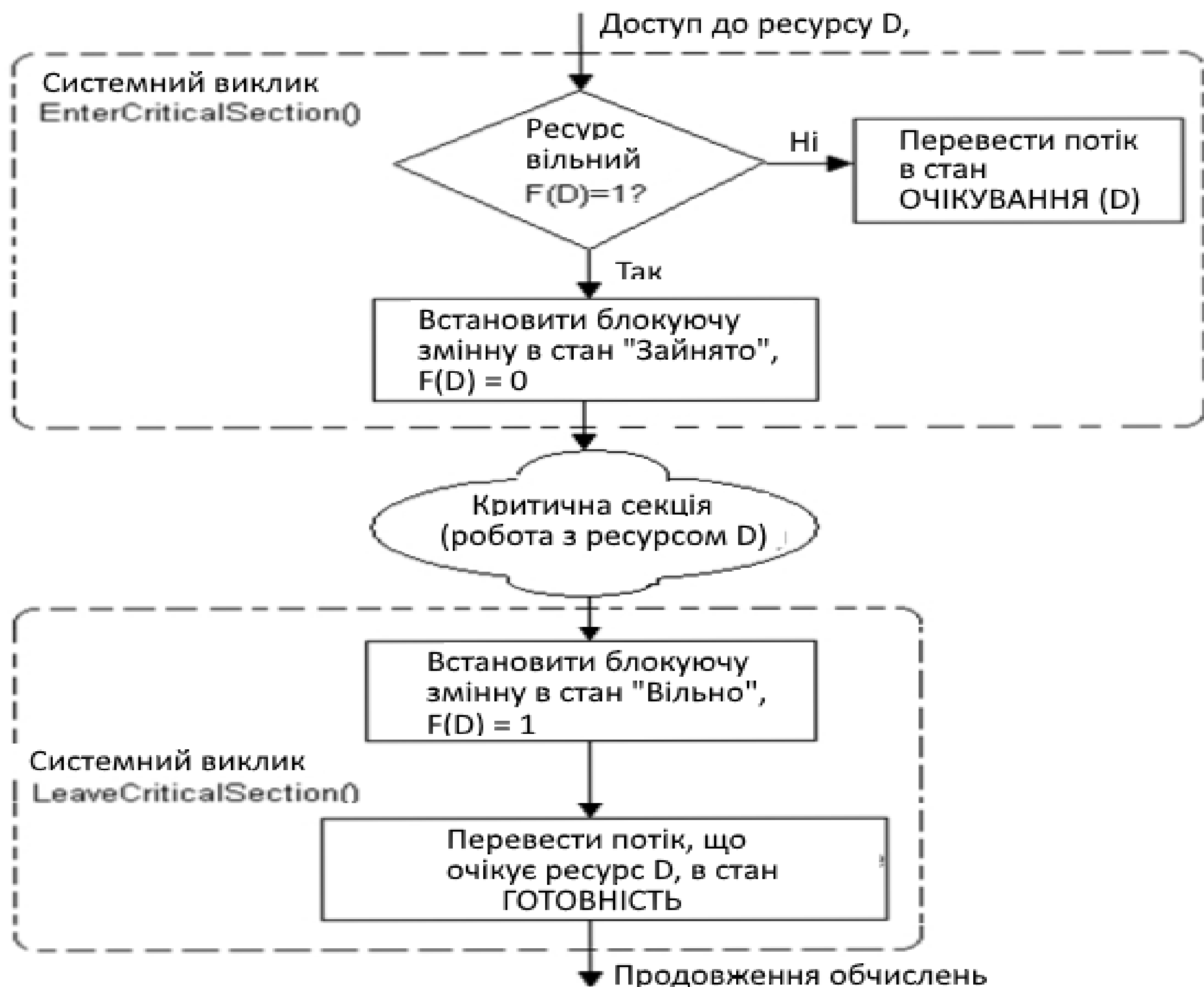


Рисунок 7.5 – Реалізація взаємного виключення з використанням системних функцій входу в критичну секцію і виходу з неї

Таким чином виключається непродуктивна втрата процесорного часу на циклічну перевірку звільнення зайнятого ресурсу.

7.4 Взаємовиключення: програмний підхід

Програмний підхід може бути реалізований для паралельних процесів, які виконуються як в однопроцесорній, так і в багатопроцесорній системі із загальною основною пам'яттю. Такі підходи припускають елементарні взаємовиключення на рівні доступу до пам'яті. Тобто, одночасний доступ (читання і/або запис) до одного і того ж елементу пам'яті упорядковується за допомогою деякого механізму. Ніякої іншої підтримки з боку апаратного забезпечення ОС або мови програмування не передбачається.

7.4.1 Алгоритм Деккера

Дейкстра представив алгоритм взаємних виключень для двох процесів, запропонований голландським математиком Деккером (1965 р.). Алгоритм Деккера є першим відомим точним рішенням взаємного виключення без заборони переривань. Назва алгоритму пов'язана з голландським математиком Теодором Деккером, який розв'язав цю проблему. Алгоритм дозволяє двом потокам спільно використати одноразовий ресурс без конфліктів, використовуючи для зв'язку лише загальну пам'ять (лістинг 7.3). Детально з алгоритмом можна познайомитися в роботі [12].

Листинг 7.3. Алгоритм Деккера для двох процесів

```
 1  static int right = 0;
 2  static char wish[2] = { 0,0 };
 3  void csBegin ( int proc ) {
 4   int competitor;
 5   competitor = other ( proc );
 6   while (1) {
 7     wish[proc] = 1;
 8     do {
 9      if ( ! wish[competitor] ) return;
10        }
11     while ( right != competitor );
12     wish[proc] = 0;
13     while ( right == competitor );
14     }
15 }
16 void csEnd ( int proc ) {
17   right = other ( proc );
18   wish[proc] = 0;
19 }
```

Алгоритм передбачає, по-перше, загальну змінну right для представлення номера процесу, який має переважне (але не абсолютне) право на вхід в критичну секцію. По-друге, масив wish, кожен елемент якого відповідає одному з процесів і представляє «бажання» процесу увійти до критичної секції. Процес заявляє про своє «бажання» увійти до секції (рядок 7). Якщо при цьому з'ясовується, що процес-конкурент не виставив свого «бажання» (рядок 9), то відбувається

повернення з функції, тобто, процес входить в критичну секцію незалежно від того, кому належало переважне право на вхід.

Якщо ж в рядку 9 з'ясовується, що конкурент теж виставив «бажання», то перевіряється право на вхід (рядок 10). Якщо право належить нашому процесу, то повторюється перевірка «бажання» конкурента (рядки 8-10), поки воно не буде скасовано. Конкурент змушений буде відмінити своє «бажання», тому що він в цій ситуації перейде до рядка 11, де процес, що не має переважного права, повинен це зробити. Після відміни свого «бажання» процес чекає, поки переважне право не повернеться до нього (рядок 12), а потім знову повторює заяву «бажання» і так далі (рядки 6-13). Таким чином, процес у функції csBegin або повторює цикл 7-14, або виходить з функції і входить в критичну секцію (10). При виході з критичної секції (функція csEnd) процес передає переважне право входу конкурентові (рядок 16) і відмовляється від свого «бажання» (рядок 17).

7.4.2 Алгоритм Петерсона

Алгоритм Деккера розв'язує задачу взаємних виключень, але досить складно, і на додаток важко довести його коректність. У 1981 році Петерсон (Peterson) запропонував витонченіше і простіше вирішення проблеми взаємних виключень, яке перевело алгоритм Деккера в розряд застарілих.

Узагальнений алгоритм Петерсона [15] для двох процесів наведений на лістингу 7.4.

Листинг 7.4. Алгоритм Петерсона для двох процесів

```
1  static int right;
2  static char wish[2] = { 0,0 };
3  void csBegin ( int proc ) {
4      int competitor;
5      if ( proc == 0 ) competitor = 1;
6      else competitor = 0;
7      wish[proc] = 1;
8      right = competitor;
9      while (wish[competitor] && (right == competitor);
10 }
11 void csEnd ( int proc ) {
12     wish[proc] = 0;
13 }
```

При вході в критичну секцію процес заявляє про своє «бажання» (рядок 7) і відмовляється від свого переважного права (рядок 8). Процес чекатиме, якщо його конкурент заявив своє «бажання» і має переважне право (рядок 9). Якщо немає інтересу конкурента або якщо незалежно від інтересу конкурента наш процес має переважне право, то процес входить в критичну секцію.

Якщо наш процес відмовився від свого права в рядку 8, то право нашого процесу може бути відновлене конкурентом, коли останній теж увійде до функції csBegin свого коду і виконає рядок 8. При виході з критичної секції процес просто знімає свій признак входу, і тоді його конкурент, очікуючий в рядку 8, одержує можливість виходу з циклу рядка 9 за першою частиною умови.

Узагальнення алгоритму Деккера для N процесів наведене в роботі [13].

Загальні позитивні властивості алгоритмів, що грунтуються на неальтернативних перемикачах (Деккера і Петерсона), такі:

- вони коректні як для одно-, так і для багатопроцесорних систем;
- вони ліберальні, оскільки дозволяють швидшим процесам входити у свої критичні секції частіше, ніж повільним;
- вони не обмежують кількість обслуговуваних ними процесів;
- вони дозволяють процесам скільки завгодно довго затримуватися поза своєю критичною секцією.

Але, запропоновані алгоритми, мають і недоліки:

- рішення непрості для розуміння і помилитися в їх реалізації дуже легко;
- процеси використовують зайняте очікування при вході в критичну секцію.

7.4.3 Алгоритм Лемпорта

Алгоритм Лемпорта (алгоритм булочної, Lamport's bakery algorithm) розв'язує задачу взаємного виключення для *N* процесів як для багатопроцесорних, так і розподілених систем обробки даних. Він був розроблений Леслі Лемпортом і вперше описаний в статті [16]. Остаточна редакція алгоритму вийшла в 2001 році в роботі [12].

Основна ідея алгоритму запозичена з принципу роботи магазину. Процеси (по аналогії з покупцями) вибирають собі номери, а потім процес, що має найменший номер, входить в критичну секцію.

Розглянемо цей метод детальніше. Уявимо собі булочну, де при вході стоїть автомат, який кожному клієнтові, що приходить, видає листочок з написаним на ньому номером. При цьому номер збільшується з приходом кожного нового клієнта. Кожного разу, коли продавець виявляється вільним, він обслуговує клієнта з найменшим номером. Наведемо приклад можливої послідовності дій.

Новий клієнт (A) заходить у булочну. Автомат видає йому номер 1.

Новий клієнт (B) заходить у булочну. Автомат видає йому номер 2.

Продавець звільняється і обслуговує клієнта A (у цей момент у булочній знаходяться клієнти з номерами 1 і 2, найменший з цих номерів – один).

Новий клієнт (C) заходить у булочну. Автомат видає йому номер 3.

Продавець звільняється і обслуговує клієнта B (у цей момент у булочній знаходяться клієнти з номерами 2 і 3, найменший з цих номерів – два).

Продавець звільняється і обслуговує клієнта C (у цей момент у булочній знаходиться тільки клієнт з номером 3).

Новий клієнт (D) заходить у булочну. Автомат видає йому номер 4.

Продавець звільняється і обслуговує клієнта D (у цей момент у булочній знаходиться тільки клієнт з номером 4).

Новий клієнт (E) заходить у булочну. Автомат видає йому номер 5.

Новий клієнт (F) заходить у булочну. Автомат видає йому номер 6.

Новий клієнт (G) заходить у булочну. Автомат видає йому номер 7.

Продавець звільняється і обслуговує клієнта E.

Продавець звільняється і обслуговує клієнта F.

Новий клієнт (H) заходить у булочну. Автомат видає йому номер 8.

Продавець звільняється і обслуговує клієнта G.

Продавець звільняється і обслуговує клієнта H.

У цій моделі усі дії відбувалися послідовно. У реальності ж може статися так, що два клієнти прийдуть одночасно. Які номери їм видати в цьому випадку? Одне з можливих рішень цієї проблеми може бути таким: видамо їм однакові номери, і дамо продавцеві вказівку при виборі з двох клієнтів з однаковим номером обслуговувати того клієнта, в якого, наприклад, менше номер паспорта. Таким чином, вважається, що усі клієнти якось впорядковані за пріоритетом.

Алгоритм Лемпорта діє аналогічним чином. При цьому роль клієнтів грають потоки, а обслуговування продавцем відповідає діям процесу в критичній секції. Аналогом номера паспорта в разі взаємодії потоків служитиме ідентифікатор потоку. Наведемо реалізацію цього алгоритму на псевдокоді (лістинг 7.5), в тому вигляді, як він наведений в книзі з паралельного і розподіленого програмування [4].

Листинг 7.5. Алгоритм Лемпорта перевірки взаємних виключень

```
1 void lock(int i) {
2   choosing[i] = true;
3   for (int j = 0; j < n; j++) {
4     if (number[j] >= number[i])
5       number[i] = number[j];
6     }
7     number[i]++;
8     choosing[i] = false;
9     for (int j = 0; j < n; j++) {
10       while (choosing[j]);
11       while ((number[j] <> 0) && ((number[j] < number[i])
12       || ((number[j] == number[i]) && (j < i))));
13     }
14 }
1 void unlock(int i) {
2 number[i] = 0;
3 }
```

Наведений запис алгоритму важкий для розуміння. З детальнішим описом алгоритму можна познайомитися в роботах [12; 18].

7.5 Взаємне виключення: апаратна підтримка

Взаємне виключення за допомогою змінних-перемикачів базується на атомарності звернень до пам'яті. Як було показано вище, це робить рішення універсальним як для одно-, так і для багатопроцесорних систем. Але більшість архітектур комп'ютерів мають у складі своєї системи спеціальні команди з розширеною атомарністю звернень до пам'яті, за допомогою яких можна реалізувати взаємне виключення.

Якщо в системі є тільки один процесор, то паралельні процеси не можуть перекриватися, а здатні тільки чергуватися. Крім того, процес триватиме до тих пір, поки не буде викликаний сервіс операційної системи або доки процес не буде перерваний. Тому, для того щоб гарантувати взаємне виключення, **досить захистити процес від переривання**. Процес у такому разі може забезпечити взаємне виключення таким чином:

```
While (true)  {
        /* Заборона переривань */
        /* Критичний розділ */
        /* Дозвіл переривань */
        /* Інший код */
}
```

Оскільки робота програми в критичнному розділі не може бути перервана (неможливе переривання навіть по таймеру), виконання взаємного виключення гарантується. Проте ціна такого підходу висока. Ефективність роботи може помітно знизитися, оскільки при цьому обмежена можливість процесора з чергування програм. Було б також безрозсудно давати призначеному для користувача процесу можливість дозволу і заборони переривань в усій обчислювальній системі. Уявіть собі, що буде, якщо процес заборонив усі переривання і в результаті якого-небудь збою не включив їх назад. ОС на цьому може закінчити своє існування. Інша проблема полягає в тому, що такий підхід не працюватиме в багатопроцесорній архітектурі, оскільки одночасно може працювати декілька процесів. У цьому випадку заборона переривань не гарантує виконання взаємовиключень.

Як уже згадувалося, на рівні апаратного забезпечення звернення до елементу пам'яті виключає будь-які інші звернення до цього елементу. Грунтуючись на цьому принципі, розробники процесорів пропонують ряд машинних команд, які за один цикл вибірки команди ***атомарно*** виконують над елементом пам'яті дві дії, такі як читання і запис, або читання і перевірка значення. Оскільки ці дії виконуються в одному циклі, на них не в змозі вплинути ніякі інші інструкції.

Головним недоліком описаної технології є те, що потік повинен постійно перевіряти в циклі, чи не зняли блокування. Таку ситуацію називають ***активним очікуванням***, а таке блокування – ***спін-блокуванням*** (spinlock). Розглянемо дві з інструкцій (команд), що найчастіше реалізовуються.

7.5.1 Команда TestAndSet

Інструкцію перевірки і установки значення можна визначити як:

```
Boolean TestAndSet(int I) {
        If I == 0) {
          I = 1;
          Return true;
        }
        else return alse;
}
```

Інструкція (команда) TestAndSet (перевірити і присвоїти 1) перевіряє значення свого аргументу *I*. Якщо він дорівнює 0, функція замінює його на 1 і повертає *true*. Інакше значення змінної не міняється і повертається *false*. Функція *TestAndSet* виконується атомарно, тобто її виконання не може бути перерване.

У лістингу 7.6 показаний протокол взаємних виключень, який грунтується на використанні описаної інструкції.

Лістинг 7.6. Апаратна перевірка взаємних виключень. Інструкція перевірки і установки:

```
const int n = /* Кількість процесів */;
int bolt;
void P(int I)  {
  while(true)      {
        while(!testset(bolt)); /* чекати */
        /* Критичний розділ */
        bolt = 0;
        /* інша частина коду */
  }
}
void main() {
        bolt = 0;
        parbegin(P(1),P(2),…,P(n));
}
```

Змінна *bolt,* що розділяється, ініціалізувалася нульовим значенням. Тільки процес, який може увійти до критичного розділу, знаходить, що значення змінної *bolt* дорівнює 0. Усі інші процеси при спробі входу в критичний розділ переходять в режим очікування. Вийшовши із критичного розділу, процес встановлює заново значення *bolt* рівним 0, після чого знову тільки один процес з очікуючих входу в критичний розділ отримає потрібний йому доступ. Вибір цього процесу залежить від того, якому з процесів вдалося виконати інструкцію *testset* першим.

7.5.2 Команда обміну

Виконання команди Swap (обміняти значення регістра і елементу пам'яті), що обмінює два значення, які знаходяться в пам'яті, можна проілюструвати такою функцією:

```
Void Swap (int register, int memory) {
        int temp;
        temp = memory;
        memory = register;
        register = temp;
}
```

У процесі її виконання доступ до елементу пам'яті для всіх інших процесів блокується.

У лістингу 7.7 показаний протокол взаємних виключень, заснований на використанні описаної інструкції. Змінна *bolt,* що розділяється, ініціалізувалася нульовим значенням. У кожного процесу є локальна змінна *key,* що ініціалізувалася значенням 1. У критичний розділ може увійти тільки один процес, який виявляє, що значення змінної *bolt* дорівнює 0. Цей процес забороняє вхід в критичний розділ усім іншим процесам шляхом установки значення *bolt*, рівним 1. Вийшовши з критичного розділу процес заново встановлює значення *bolt*, рівним 0.

Лістинг 7.7. Апаратна перевірка взаємних виключень. Інструкція обміну

```
const int n = /* Кількість процесів */;
int bolt;     /* змінна, що розділяється, */
void P (int I) {
 int keyi;
 while(true){
       keyi = 1;
       while (keyi != 0)
              Swap (keyi, bolt);
       /* Критичний розділ */
       Swap (keyi, bolt);
       /* інша частина коду */
 }
}
void main () {
bolt = 0;
parbegin(P(1),P(2),…,P(n));
}
```

Якщо *bolt = 0*, то в критичному розділі немає жодного процесу. Якщо *bolt = 1*, то в критичному розділі знаходиться рівно один процес, а саме той, змінна *key* якого має нульове значення.

Контрольні питання і тести до розділу 7

Контрольні питання

1. Чи зменшується швидкість роботи процесу, якому відмовлено в негайному доступі до ресурсу?
2. Чи залежить результат виконання потоків від послідовності виконання операцій потоків в системі?
3. Як називається ситуація, коли два (і більше) потоки (процеси) прочитують або записують дані одночасно, і кінцевий результат залежить від того, який з них був першим.
4. Дайте визначення поняття «Критичного ресурсу» і частини програми, яка його використовує, – «критичному розділу»
5. Як називається ситуація, коли в критичній секції, пов'язаній з яким-небудь ресурсом, у будь-який момент часу може знаходитися тільки один процес?
6. Який алгоритм взаємних виключень для двох процесів представив Дейкстра, який є першим відомим точним рішенням взаємного виключення без заборони переривань?

7. Хто запропонував витонченіше і простіше рішення проблеми взаємних виключень, яке перевело алгоритм Деккера в розряд застарілих?
8. Як ще назвают алгоритм Лемпорта?
9. Принцип роботи двох інструкцій (команд) апаратної перевірки взаємних виключень.
10. Яким чином написання програмного коду критичної секції може призвести до ситуації нескінченного відкладання і навіть до взаємоблокування?

Тести

1. При спільному використанні процесами апаратних і інформаційних ресурсів обчислювальної системи виникає потреба в:
    1) синхронізації;
    2) адаптації;
    3) оптимізації;
    4) буферизації.
2. Яка операція називається атомарною?
    1) проста машинна команда;
    2) операція, що виконується одним процесом;
    3) неподільна операція, що виконується без переривання діяльності;
    4) операція, що виконується за один машинний такт центрального процесора.
3. Термін «критична секція» належить:
    1) до ділянки коду процесу з найбільшим об'ємом обчислювальної роботи;
    2) до ділянки коду процесу, в якому процес спільно з іншими процесами використовує змінні, що розділяються;
    3) до ділянки коду процесу, виконання якого спільне з іншими процесами може призвести до неоднозначних результатів.
4. Що таке виділений ресурс?
    1) пристрій, що монопольно використовується процесом;
    2) пристрій або дані, до яких процес має ексклюзивний доступ;
    3) дані, що заблоковані процесом для виняткового доступу.
5. Термін race condition (умова гонки) належить:
    1) до набору процесів, що спільно використовують який-небудь ресурс;
    2) до набору процесів, що демонструють недетерміновану поведінку;
    3) до набору процесів, для кожного з яких важливо завершитися якнайшвидше.
6. Засіб обчислювальної системи, який може бути виділений процесу на певний інтервал часу, називається:
    1) потоком;
    2) перериванням;
    3) системним викликом;
    4) ресурсом.

8 СЕМАФОРИ, МОНІТОРИ І ПОВІДОМЛЕННЯ

Розглянуті раніше алгоритми синхронізації хоча і є коректними, але досить громіздкі. Більше того, процедура очікування входу в критичну ділянку, як для алгоритму Петерсона, так і при використанні апаратних інструкцій, припускає досить тривале обертання процесу в порожньому циклі, тобто марну витрату дорогоцінного часу процесора, перевіряючи можливість увійти до критичної області.

Існують і інші серйозні недоліки алгоритмів, побудованих засобами звичайних мов програмування. Припустимо, що в обчислювальній системі знаходяться два взаємодіючі процеси: один з них – *H* з високим пріоритетом, інший – *L* з низьким пріоритетом. Нехай планувальник влаштований так, що процес з високим пріоритетом витісняє низькопріоритетний процес всякий раз, коли той готовий до виконання, і займає процесор на весь відведений йому час (якщо не з'явиться процес з ще більшим пріоритетом). Тоді в разі, якщо процес *L* знаходиться у своїй критичній секції, а процес *H,* отримавши процесор, підійшов до входу в критичну область, ми отримуємо тупикову ситуацію. Процес *H* не може ввійти до критичної області, знаходячись в циклі, а процес *L* не отримує управління, щоб покинути критичну ділянку.

Для того щоб не допустити виникнення подібних проблем, були розроблені різні механізми синхронізації вищого рівня. Опису ряду з них – семафорів, моніторів і повідомлень – і присвячений цей розділ.

8.1 Семафори

Першою великою роботою, присвяченою питанням паралельних обчислень, стала монографія Дейкстри, яка була видана в 1965 році [14]. У ній розглядалася розробка ОС як побудова певної кількості співпрацюючих послідовних процесів і створення ефективних та надійних механізмів підтримки цієї співпраці. Ці механізми легко застосовуються і призначеними для користувача процесами, якщо процесор і ОС роблять їх загальнодоступними.

Дейкстра запропонував узагальнений засіб синхронізації процесів, ввівши два нових примітиви. В абстрактній формі ці примітиви, позначені Дейкстрою як *P i V* (це перші букви голландських слів перевірка (Proberen) і збільшення (Verhogen)), оперують над цілими не від'ємними змінними *S,* що називаються ***семафорами*** (semaphore).

Класичне визначення цих операцій виглядає таким чином:

P(S): доки S = 0 процес блокується;

S = S - 1;

V(S): S = S + 1;

Позначимо ці примітиви відповідно як *Wait i Signal* (або *Down(s)* і *Up(s)*, як це було ще в нотації Algol-68).

Для передачі сигналу через семафор *S* процес застосовує примітив *Signal(s),* а для отримання сигналу – примітив *Wait(s)*. В останньому випадку процес призупиняється до тих пір, поки не буде прийнятий відповідний сигнал.

Над семафором визначені три операції, що наводяться нижче.

1. Семафор може ініціалізуватися тільки додатнім числом.
2. Операція *Wait(s)* зменшує значення семафора на 1. Якщо це значення стає негативним, то процес блокується, тобто чекає, поки це зменшення стане можливим. Успішна перевірка, зменшення і перехід процесу в стан очікування виконуються як єдина і неподільна елементарна (атомарна) дія.
3. Операція *Signal(s)* збільшує значення семафора на 1 однією неподільною дією (вибірка, інкремент і запам'ятовування). Якщо це значення від'ємне, то заблокований операцією *Wait* процес деблокується. Тобто, один з процесів вибирається системою і йому дозволяється завершити свою операцію *Wait(s).* Операція збільшення значення семафора і активізації процесу також неподільна. Є різні організації семафорів щодо операції *Signal(s)*, застосованої до семафора, пов'язаного з декількома очікуючими процесами, значення семафора так і залишається рівним 0, але число очікуючих процесів зменшується на одиницю.

Таким чином, якщо значення семафора більше нуля, операція *Wait(s)* зменшує його і просто повертає управління. Якщо значення дорівнює нулю, процедура *Wait(s)* не повертає управління процесу, а процес переводиться в стан очікування.

При виконанні операції *Signal(s),* якщо з цим семафором пов'язані один або декілька процесів, які не можуть завершити ранішу операцію *Wait(s),* один з них вибирається системою і йому дозволяється завершити свою операцію *Wait(s).* Таким чином, після операції *Signal(s)*, що застосовується до семафора, пов'язаного з декількома очікуючими процесами, значення семафора так і залишається рівним нулю, але число очікуючих процесів зменшується на одиницю. Жоден процес не може бути блокований під час виконання операції *Signal(s).*

В окремому випадку, коли семафор *S* може набувати тільки значень 0 і 1, він перетворюється на блокуючу змінну. Операція *Wait* містить в собі потенційну можливість переходу процесу, який її виконує, в стан очікування, тоді як операція *Signal* може при деяких обставинах активізувати інший процес, призупинений операцією *Wait.* Така спрощена версія семафора, називається ***м'ютексом*** (***mutex***, скорочення від **mut**ual **ex**clusion**).** У загальному випадку м'ютексом називають примітив синхронізації, який не допускає виконання деякого фрагмента коду більше як одним процесом. М'ютекс не здатний обчислювати, він може лише управляти взаємним виключенням доступу до спільно використовуваних ресурсів або коду.

Розглянемо використання семафорів на класичному прикладі взаємодії двох процесів «**письменник-читач**», що виконуються в режимі мультипрограмування, один з яких пише дані в буферний пул, а інший прочитує їх з буферного пулу.

Нехай буферний пул складається з *N* буферів, кожен з яких може містити один запис. Процес «письменник» повинен призупинятися, коли всі буфери виявляються зайнятими, і активізуватися при звільненні хоч би одного буфера. Навпаки, процес «читач» призупиняється, коли усі буфери порожні, і активізується при появі хоч би одного запису.

Введемо два семафори: *e* – число порожніх буферів і *f* – число заповнених буферів, причому в початковому стані *e =N,* а *f =0*. Припустимо, що запис у буфер і зчитування з буфера є критичними секціями. Введемо також двійковий семафор *b*, використовуваний для забезпечення взаємного виключення. Тоді робота потоків (процесів) із загальним буферним пулом може бути представлена блок-схемою (рис. 8.1).

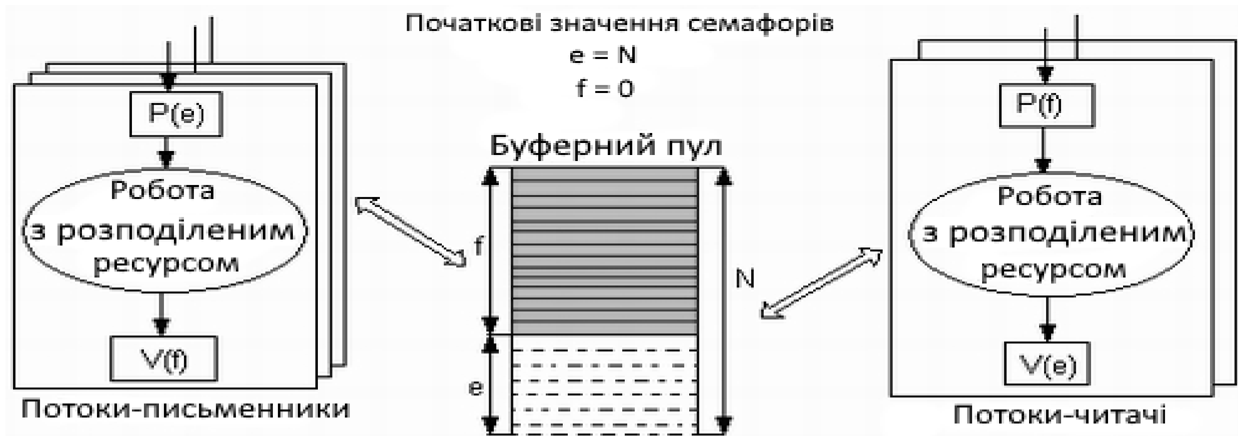


Рисунок 8.1 – Використання семафорів для синхронізації потоків

Фрагмент програми на С-подібній мові виглядає таким чином:

Лістинг 8.1. Визначення семафорних примітивів

```
// Глобальні змінні
#define N 256
int e = N, f = 0, b = 1;
void Writer () {
  while(1){
   PrepareNextRecord(); /* підготовка нового запису */
   Wait(e); /* Зменшити число вільних буферів, якщо вони є */
       /* інакше – чекати, поки вони звільняться */
   Wait(b);   /* Вхід в критичну секцію  */
   AddToBuffer();   /* Додати новий запис у буфер /
   Signal(b);    /* Вихід з критичної секції */
   Signal(f);   /* Збільшити число зайнятих буферів */
  }
}
void Reader () {
  while(1){
        Wait(f); /* Зменшити число зайнятих буферів, якщо вони есть*/
         /* інакше чекати, поки вони з'являться */
        Wait(b); /* Вхід в критичну секцію */
        GetFromBuffer(); /* Узяти запис з буфера */
        Signal(b); * Вихід з критичної секції */
        Signal(e); /* Збільшити число вільних буферів */
        ProcessRecord(); /* Обробити запис */
        }
}
```

Семафор може використовуватися і в якості блокуючої змінної. У розглянутому вище прикладі, для того щоб виключити колізії при роботі з областю пам'яті, що розділяється, запис у буфер і прочитування з буфера є критичними секціями. Взаємне виключення забезпечується за допомогою двійкового семафора ***b***. Обидва потоки після перевірки доступності буферів повинні виконати перевірку доступності критичної секції. У лістингу 8.2 наведено формальніше визначення примітивів семафорів. Передбачається, що примітиви атомарні, тобто вони не можуть бути перервані, і кожна з підпрограм розглядається як єдиний крок.

Лістинг 8.2. Визначення семафорних примітивів

```
Struct Semaphore {
      Int  count;
      QueueTypy queue;
}
void Wait(semaphore S) {
      s.count--;
      if (s.count < 0) {
        /* Помістити процес s.queue, s.count++ або =0 */
        /* Заблокувати процес */
      }
}
void Signal(semsphore S)
{
      s.count++;
      if (s.count <= 0) {
        /* Видалити процес P з s.queue */
        /* Помістити процес P в список активних */
      }
}
```

Для зберігання процесів використовується черга. При цьому виникає питання про порядок витягання з черги. Найкоректніший спосіб – використання принципу «першим увійшов – першим вийшов» (First In First Out – FIFO). При цьому з черги звільняється процес, який був заблокований довше за інших. Семафор, що використовує цей метод, називається ***сильним семафором*** (strong semaphore). Семафор, порядок витягання процесів з черги якого не визначений, називається ***слабким семафором*** (weak semaphore).

8.1.1 Взаємні виключення

Розглянемо алгоритм взаємовиключень з використанням сильного семафора *S* (лістинг 8.3), в якому гарантується неможливість голодування, але слабкий семафор такої гарантії не дає. Далі вважатимемо, що працюють сильні семафори, оскільки саме вони використовуються в ОС.

Нехай у нас є *N* процесів, що ідентифікуються масивом *P(i)*. У кожному з процесів перед входом в критичний розділ виконується виклик *Wait(S)*. Якщо значення *S* стає від'ємним, процес призупиняється. Якщо ж значення *S* дорівнює

1, воно зменшується до нуля і процес негайно входить в критичний розділ. Оскільки *S* більше не є додатнім, жоден інший процес не може ввійти до критичного розділу.

Лістинг 8.3. Взаємні виключення з використанням семафорів

```
/* Програма взаємного виключення */
const int  n = /* Кількість процесів */;
semaphore  s = 1;
void  P(int  i)
{
      while(true) {
            wait(s);
            /* Критичний розділ */
            signal(s);
            /* Інший код */
      }
}
void main() {
      perbegin(P(1),P(2),…,P(n));
}
```

Семафор ініціалізувався значенням 1. Отже, перший процес, що виконує інструкцію *Wait*, зможе негайно потрапити в критичний розділ, встановлюючи значення семафора, рівним 0. Будь-який інший процес при спробі увійти до критичного розділу виявить, що він зайнятий. Відповідно, станеться блокування процесу, а значення семафора буде зменшено до -1. Намагатися увійти до критичного розділу може будь-яка кількість процесів; кожна спроба неуспіху зменшує значення семафора.

Після того, як процес, що увійшов до критичного розділу першим, покидає його, *S* збільшується, і один із заблокованих процесів (якщо такий є) видаляється з пов'язаної з семафором черги і активується. Таким чином, як тільки планувальник ОС надасть йому можливість виконання, процес тут же зможе увійти до критичного розділу.

8.1.2 Задача «Виробник-Споживач» («Письменник-Читач»)

А тепер додамо до нашої задачі нове сховище – розглядатимемо кінцевий буфер *b(n)* як циклічне сховище, при роботі з яким значення покажчиків повинні виражатися за модулем розміру буфера. При цьому виконуються умови, що наведені нижче.

Блокування:

1. Виробник: вставка в повний буфер.
2. Споживач: видалення з порожнього буфера.

Деблокування:

1. Виробник: вставка елемента в буфер.
2. Споживач: видалення елемента з буфера.

Функції виробника і споживача при цьому можуть бути записані таким чином (*in* і *out* ініціалізувані значенням 0):

Виробник: (AddToBuffer()) **Споживач:** (GetFromBuffer())

```
Виробник:                                  Споживач:
While (true) {                             While (true) {
  /* Виробництво елементу v /;             while(in == out); /* чекати */
  while((in+1) % n == out); /*ждать*/       w = b[out];
      b[in] = v;                           out = (out+1) % n;
      in % n;                          /* споживання елементу w*/
}                                           }
```

У лістингу 8.4 наведеній розв'язок цієї задачі з використанням ***узагальнених семафорів (семафорами з лічильниками).*** Для відстежування порожнього місця в буфері в програму доданий семафор *e*.

Лістинг 8.4. Розв'зок задачі Виробник-Споживач з обмеженим циклічним буфером

```
const  int  SizeBuffer = /* Розмір буфера */;
semafore  s = 1;
semafore  n = 0;  /* число зайнятих буферів */
semafore  e = SizeBuffer; /* число порожніх буферів */
void Writer () {
      while(1){
        PrepareNextRecord(); /* підготовка нового запису */
        Wait(e); /* Зменшити число вільних буферів, якщо вони є */
    /* інакше чекати, поки вони звільняться */
        Wait(s);       /* Вхід в критичну секцію  */
        AddToBuffer();  /* Додати новий запис у буфер */
        Signal(s);     /* Вихід з критичної секції */
        Signal(n);     /* Збільшити число зайнятих буферів */
      }
}
void Reader () {
      while(1){
        Wait(n); /* Зменшити число зайнятих буферів, якщо вони є */
     /* інакше чекати, поки вони з'являться */
        Wait(s); /* Вхід в критичну секцію */
        GetFromBuffer(); /* Узяти запис з буфера */
        Signal(s);   /* Вихід з критичної секції */
        Signal(e);  /* Збільшити число вільних буферів */
        ProcessRecord(); /* Обробити запис */
      }
}
void main(){
      parbegin (Writer, Reader);
}
```

Припустимо тепер, що при переписуванні цієї програми сталася помилка: програміст переставив операції *Signal(s)* і *Signal(n)* в процедурі *Writer ().* Це призведе до того, що операція *Signal(n)* виконуватиметься в критичному розділі виробника без переривання споживача. Чи вплине це на виконання програми? Ні, споживач у будь-якому випадку повинен чекати установки обох семафорів перед продовженням роботи.

Тепер припустимо, що переставлені операції *Wait(n)* і *Wait(s)* в процедурі *Reader ()*. Це призведе до фатальних наслідків. Якщо споживач увійде до критичного розділу, коли буфер порожній (*n = 0*), то виробник не зможе додати дані в буфер і система виявиться в стані взаємного блокування.

З усього вищесказаного ясно, що використовувати семафори треба дуже обережно, і програміст при написанні програм з паралельно працюючими процесами повинен мати певну культуру програмування.

У разі потоків у призначеному для користувача просторі немає проблеми доступу потоків, наприклад до семафора, оскільки в усіх потоків загальний адресний простір. Проте, у більшості попередніх моделей, зокрема в алгоритмі Петтерсона і семафорах, передбачалося, що декілька процесів мають доступ до спільно використовуваної ділянки пам'яті. Якщо адресні простори процесів несумісні, то як вони можуть спільно використати змінну *turn* в алгоритмі Петтерсона, або семафори, або загальний буфер? На це питання існує дві відповіді.

По-перше, деякі із спільно використовуваних структур даних, скажімо, семафори, можуть зберігатися в ядрі з доступом тільки через системні запити. Цей підхід вирішує проблему.

По-друге, більшість сучасних ОС (Windows, Unix) представляють можливість спільного використання процесами деякої частини адресного простору. У цьому випадку можливе розділення буфера і інших структур даних. У крайньому випадку можна використати файл.

8.1.3 Java семафори

Семафори представляють ще один засіб синхронізації для доступу до ресурсу. У Java семафори представлені класом **Semaphore** з пакету *java.util.concurrent*.

Для управління доступом до ресурсу семафор використовує лічильник, що представляє кількість дозволів. Якщо значення лічильника більше нуля, то потік дістає доступ до ресурсу, при цьому лічильник зменшується на одиницю. Після закінчення роботи з ресурсом потік звільняє семафор, і лічильник збільшується на одиницю. Якщо ж лічильник дорівнює нулю, то потік блокується і чекає, поки не отримає дозвіл від семафора.

Встановити кількість дозволів для доступу до ресурсу можна за допомогою конструкторів класу Semaphore:

```
1   Semaphore(int permits)
2   Semaphore(int permits, boolean fair)
```

Параметр *permits* вказує на кількість допустимих дозволів для доступу до ресурсу. Параметр *fair* у другому конструкторі дозволяє встановити черговість отримання доступу. Якщо він рівний true, то дозволи надаватимуться очікуючим потокам в тому порядку, в якому вони просили доступ. Якщо ж він рівний *false,* то дозволи надаватимуться в невизначеному порядку.

Для отримання дозволу в семафора потрібно викликати метод **acquire()**, який має дві форми:

```
1   void acquire() throws InterruptedException
2   void acquire(int permits) throws InterruptedBxception
```

Для отримання одного дозволу застосовується перший варіант, а для отримання декількох дозволів – другий варіант.

Після виклику цього методу доки потік не отримає дозвіл, він блокується. Після закінчення роботи з ресурсом отриманий раніше дозвіл потрібно звільнити за допомогою методу **release()**:

```
1   void release()
2   void release(int permits)
```

Перший варіант методу звільняє один дозвіл, а другий варіант – кількість дозволів, вказаних в *permits*. Використаємо семафор Java в простому прикладі (лістинг 8.5).

Лістинг 8.5. Приклад використання семафорів у Java.

```
1    import java.util.concurrent.Semaphore;
2
3    public class ThreadsApp {
4
5      public static void main(String[] args) {
6
7          Semaphore sem = new Semaphore(1); // 1 дозвіл
8          CommonResource res = new CommonResource();
9          new Thread(new CountThread(res, sem, "CountThread 1")).start();
10         new Thread(new CountThread(res, sem, "CountThread 2")).start();
11         new Thread(new CountThread(res, sem, "CountThread 3")).start();
12     }
13   }
14   class CommonResource{
15
16       int x=0;
17   }
18
19   class CountThread implements Runnable{
20
21     CommonResource res;
22     Semaphore sem;
23     String name;
24     CountThread(CommonResource res, Semaphore sem, String name){
25        this.res=res;
26        this.sem=sem;
27        this.name=name;
28     }
29
30     public void run(){
31
32       try{
33         System.out.println(name + " чекає дозвіл");
34         sem.acquire();
35         res.x=1;
```

```
36          for (int i = 1; i < 5; i++){
37            System.out.println(this.name + ": " + res.x);
38            res.x++;
39            Thread.sleep(100);
40          }
41        }
42        catch(InterruptedException e)
43          {System.out.println(e.getMessage());}
44        System.out.println(name + " звільняє дозвіл");
45        sem.release();
46      }
47    }
```

У даному прикладі є загальний ресурс *CommonResource* з полем *x*, яке змінюється кожним потоком. Потоки представлені класом *CountThread*, який отримує семафор і виконує деякі дії над ресурсом. В основному класі програми (*main*) ці потоки запускаються. У результаті ми отримаємо такі повідомлення на екрані:

```
CountThread 1 чекає дозвіл
CountThread 2 чекає дозвіл
CountThread 3 чекає дозвіл
CountThread 1: 1
CountThread 1: 2
CountThread 1: 3
CountThread 1: 4
CountThread 1 звільняє дозвіл
CountThread 3: 1
CountThread 3: 2
CountThread 3: 3
CountThread 3: 4
CountThread 3 звільняє дозвіл
CountThread 2: 1
CountThread 2: 2
CountThread 2: 3
CountThread 2: 4
CountThread 2 звільняє дозвіл
```

Семафори чудово підходять для розв'зання задач, де потрібно обмежувати доступ. Наприклад, класична задача про філософів, що обідають, яка використовується в інформатиці для ілюстрації проблем синхронізації при розробці паралельних алгоритмів. Задача була сформульована в 1965 році Едсгером Дейкстрою як екзаменаційна вправа для студентів. Як приклад був узятий конкуруючий доступ до стрічкового накопичувача. Незабаром задача

була сформульована Річардом Хоаром в тому вигляді, в якому вона відома сьогодні. Суть її полягає в наступному.

П'ять безмовних філософів сидять навколо круглого столу, перед кожним філософом стоїть тарілка спагеті. Вилки лежать на столі між кожною парою найближчих філософів. Кожен філософ може або їсти, або роздумувати (сидячи за столом або ходити). Їда не обмежена кількістю спагеті – мається на увазі нескінченний запас. Проте, філософ може їсти тільки тоді, коли тримає дві вилки – узяту справа і ліворуч.

Кожен філософ може узяти найближчу вилку (якщо вона доступна), або покласти, якщо він вже тримає її. Узяття кожної вилки і повернення її на стіл є роздільними діями, які повинні виконуватися одне за іншим.

Суть проблеми полягає в тому, щоб розробити модель поведінки (паралельний алгоритм), при якій жоден з філософів не голодуватиме, тобто вічно буде чередувати їжу і роздуми.

Задача сформульована так, щоб ілюструвати проблему уникнення взаємного блокування – стану системи, при якому прогрес неможливий. Відносно просте рішення задачі досягається шляхом додавання офіціанта біля столу. Філософи повинні чекати дозволу офіціанта перед тим, як узяти вилку. Оскільки офіціант знає скільки вилок використовується в даний момент, то він може приймати рішення відносно розподілу вилок і тим самим запобігти взаємного блокування філософів. Якщо чотири вилки з п'яти вже використовуються, то наступний філософ, що запросив вилку, змушений буде чекати дозволу офіціанта, який не буде отримано, поки вилка не буде звільнена. Передбачається, що філософ завжди намагається спочатку узяти ліву вилку, а потім – праву (чи навпаки), що спрощує логіку. Офіціант працює, як семафор.

Лістинг 8.6. Задача про обідаючих філософів.

```
1    import java.util.concurrent.Semaphore;
2
3    public class ThreadsApp {
4
5        public static void main(String[] args) {
6
7          Semaphore sem = new Semaphore(2);
8          for(int i=1;i<6;i++)
9             new Philosopher(sem,i).start();
10       }
11   }
12   // клас філософа
13   class Philosopher extends Thread
14   {
15     Semaphore sem; // семафор. що обмежує число філософів
16     // кількість їди
17     int num = 0;
18     // умовний номер філософа
19     int id;
20     // в якості параметрів конструктора передаємо
```

```
21      // ідентифікатор філософа і семафор
22      Philosopher(Semaphore sem, int id)
23      {
24        this.sem=sem;
25        this.id=id;
26      }
27
28      public void run()
29      {
30        try
31        {
32          while// доки кількість їди не досягне 3
33          {
34            //Просимо у семафора дозвіл на виконання
35            sem.acquire();
36            System.out.println("Філософ " +id+" сідає за стіл");
37            // філософ їсть
38            sleep(500);
39            num++;
40
41            System.out.println("Філософ " +id+" виходить із-за столу");
42            sem.release();
43
44            // філософ гуляє
45            sleep(500);
46          }
47        }
48        catch(InterruptedException e)
49        {
50         System.out.println("у філософа " +id+ " проблеми");
51        }
52      }
53    }
```

У результаті тільки два філософи зможуть одночасно знаходитися за столом, а інші чекатимуть:

```
Філософ 1 сідає за стіл
Філософ 3 сідає за стіл
Філософ 3 виходить із-за столу
Філософ 1 виходить із-за столу
Філософ 2 сідає за стіл
Філософ 4 сідає за стіл
Філософ 2 виходить із-за столу
Філософ 4 виходить із-за столу
Філософ 5 сідає за стіл
Філософ 1 сідає за стіл
Філософ 1 виходить із-за столу
```

```
Філософ 5 виходить із-за столу
Філософ 3 сідає за стіл
Філософ 2 сідає за стіл
Філософ 3 виходить із-за столу
Філософ 4 сідає за стіл
Філософ 2 виходить із-за столу
Філософ 5 сідає за стіл
Філософ 4 виходить із-за столу
Філософ 5 виходить із-за столу
Філософ 1 сідає за стіл
Філософ 3 сідає за стіл
Філософ 1 виходить із-за столу
Філософ 2 сідає за стіл
Філософ 3 виходить із-за столу
Філософ 5 сідає за стіл
Філософ 2 виходить із-за столу
Філософ 4 сідає за стіл
Філософ 5 виходить із-за столу
Філософ 4 виходить із-за столу
```

8.2 Монітори

Семафори забезпечують досить потужний і гнучкий інструмент для здійснення взаємних виключень і координації процесів. Але по суті, семафори – це глобальні змінні, які розділяються, що є ознакою поганої структури програми. Тому створити коректно працюючу програму з використанням семафорів не завжди легко. Складність полягає в тому, що операції *wait* і *signal* можуть бути розкидані за усією програмою, і не завжди можна відстежити їх взаємодію на контрольованому ними семафорі. Немає контролю використання семафорів і з боку компілятора або операційної системи, відповідно – немає гарантій, що семафори будуть використані правильно.

Використовувати семафори треба дуже акуратно і обережно, оскільки одна незначна помилка (наприклад, як ми вже знаємо, проста перестановка двох операцій *wait*) може призвести до зупинки системи. Це нагадує програмування на асемблері, але, насправді, ще складніше.

При використанні семафорів дві проблеми – взаємне виключення і умовна синхронізація – програмуються однією і тією ж парою примітивів, проте це різні поняття і хотілося б мати різні механізми їх реалізації.

Для того щоб спростити і полегшити роботу програмістів при написанні коректних програм, було запропоновано більш високорівневий засіб синхронізації, що називається монітором.

Монітори – це програмні модулі мови програмування, які при тій же ефективності реалізації, що і семафори, забезпечують кращу структуризацію коду. Монітор інкапсулює:

- критичні дані (змінні), що розділяються;
- функції, що реалізовують операції над даними, які розділяються;
- синхронізацію виконання паралельних потоків, що викликають вказані функції.

Потік отримує доступ до змінних у моніторі тільки шляхом виклику процедур цього монітора. Код синхронізації додається компілятором. Два потоки не можуть одночасно виконуватися в одному моніторі, оскільки не можуть одночасно виконуватися дві процедури монітора. Тобто, тільки один процес може бути активним стосовно монітора (рис. 8.2).

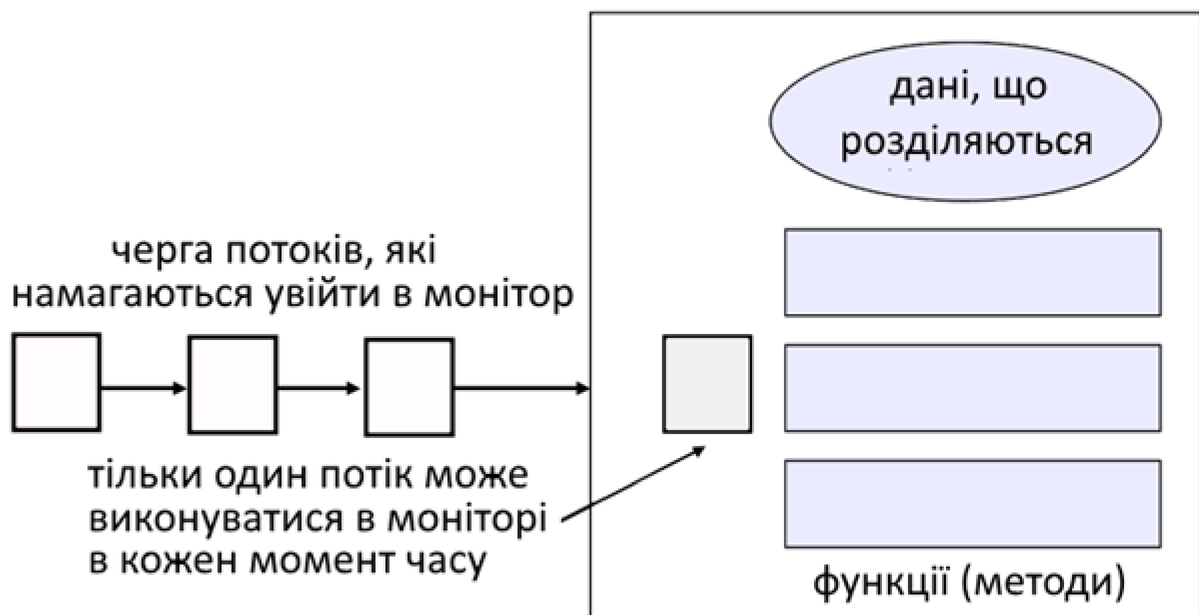


Рисунок 8.2 – Проста модель монітора

Компілятор обробляє виклики процедур монітора особливим чином. Коли процес викликає процедуру монітора, то перші декілька інструкцій цієї процедури перевіряють, чи не активний який-небудь інший процес стосовно цього монітора. Якщо так, то процес призупиняється, поки інший процес не звільнить монітор. Таким чином, **виключення входу декількох процесів в монітор реалізується не програмістом, а компілятором**, що робить помилки менш імовірними.

Уперше формальне визначення концепції монітора з деякими відмінностями було дане в 1974 році **Хоаром** (Hoare) і **Бринч Хансеном** (Brinch Hansen). Розглянемо монітор версії Хоара.

8.2.1 Монітори з сигналами (монітори Хоара)

Монітор являє собою програмний модуль, що складається з ініціюючих послідовностей, однієї або декількох процедур і локальних даних (умовних

змінних) Хоча в різних мовах монітори оголошуються і створюються по-різному, будемо вважати монітор статичним об'єктом виду:

```
monitor Monname{
        [оголошення постійних змінних]
        [оператори ініціалізації]
        [процедури]
}
```

Основні його характеристики:

1. Локальні змінні монітора доступні тільки його процедурам.
2. Процес входить в монітор шляхом виклику однієї з його процедур.
3. У моніторі в певний момент часу може виконуватися тільки один процес. Будь-який інший процес, що викликає монітор, буде призупинений в очікуванні доступності монітора.

Перші дві характеристики примушують нас згадати про об'єкти в ООП. Фактично об'єктно-орієнтовані мови програмування можуть легко організувати монітор як об'єкт із спеціальними характеристиками.

Дотримання умови виконання тільки одного процесу в певний момент часу дозволяє монітору забезпечити взаємовиключення. Дані монітора доступні в цей момент тільки одному процесу. Отже, захистити спільно використовувані структури даних можна, просто помістивши їх в монітор. Якщо дані в моніторі представляють деякий ресурс, то монітор забезпечує взаємовиключення при зверненні до ресурсу.

На абстрактному рівні можна описати структуру монітора таким чином:

```
monitor monitor_name {
  <опис внутрішніх змінних>;
  void m1(...){...
  }
  void m2(...){...
  }
  ...
  void mn(...){...
  }
  {
   < блок ініціалізації внутрішніх змінних>;
  }
}
```

Тут функції $m_1,...,m_n$ є функціями-методами монітора, а блок ініціалізації внутрішніх змінних містить операції, які виконуються один і тільки один раз: при створенні монітора або при найпершому виклику якої-небудь функції-методу до її виконання.

Для широкого застосування в паралельних обчисленнях монітори повинні включати елементи синхронізації. Припустимо, що процес, знаходячись в моніторі, має бути призупинений до виконання деякої умови. При цьому потрібно мати деякий механізм, який не лише призупиняє процес, але і звільняє

монітор для інших процесів. Пізніше, коли умова виявиться виконаною, а монітор доступним, призупинений процес зможе продовжити свою роботу з того місця, де він був призупинений.

Монітор підтримує синхронізацію за допомогою змінних умови, розташованих (і доступних) тільки в моніторі. Працювати з цими змінними можуть дві функції:

1. *cwait(c)* – призупиняє виконання процесу по умові *c*.
2. *csignal(c)* – поновлює виконання деякого процесу, призупиненого викликом *cwait(c)* з тією ж умовою. Якщо є декілька таких процесів, вибирається один з них. Якщо таких процесів немає, функція не робить нічого.

Зверніть увагу на те, що операції *wait/signal* монітора відрізняються від відповідних операцій семафора. Якщо процес в моніторі передає сигнал, але при цьому немає жодного очікуючого його процесу, то сигнал втрачається.

Якщо один з процесів увійшов до монітора, то інші процеси, які намагаються увійти до монітора, приєднуються до черги процесів, призупинених в очікуванні доступності монітора. Якщо ж процес в моніторі тимчасово зупиняється, виконавши виклик *cwait(x)*, то процес поміщається в чергу процесів, які очікують повторного входу в монітор при виконанні умови.

Якщо процес, що виконується в моніторі, виявить друге значення змінної умови *x,* то він виконує операцію *csignal(x)*, яка повідомляє про виявлену зміну відповідної черги. Як приклад використання монітора повернемося до задачі «Виробник-Споживач» з обмеженим буфером. У лістингу 8.7 показаний розв'язок цієї задачі з використанням монітора. Модуль монітора *ProducerConsumer* управляє буфером для зберігання і отримання символів. Монітор включає дві змінні умов: *notfull* істинно, якщо в буфері є місце хоч би для одного символу, а *notempty* істинно, якщо в буфері є хоч би один символ.

Лістинг 8.7. Розв'язок задачі «Виробник-Споживач» з обмеженим буфером з використанням монітора Хоара

```
Monitor ProducerConsumer;
Char buffer [N]          /* Місце для N елементів */
Int nextin, nextout;     /* Покажчиків буфера */
Int count;               /* Кількість елементів у буфері */
Int notfull, notempty;   /* Синхронізація */
Void Append (char x) {
       if (count == N)
              cwait(notfull);     /* Буфер заповнений */
       Buffer [nextin] = x;
       Nextin % N;
       Count++;          /* Додаємо елемент у буфер */
       Csignal(notempty);     /* Відновлення роботи споживача */
}
Void Take (char x) {
       if (count == 0)
              cwait(notempty);/* Буфер порожній */
       x = Buffer [nextout];
```

```
        Nextout % N;
        Count--;    /* Видаляємо елемент з буфер */
        Csignal(notfull); /* Поновлюємо роботу виробників */
}
{       /* Тіло монітора */
        nextin = 0;/* Спочатку буфер порожній */
        nextout = 0;
        count = 0;
}
Void Producer () {
        char x;
        While (true) {
                Produce (x);
                Append (x);
        }
}
Void Consumer () {
        char x;
        While (true) {
                Take (x);
                Consume (x);
        }
}
Void Main () {
        Perbegin (Producer, Consumer);
}
```

Виробник може додавати символи в буфер тільки з монітора за допомогою процедури Append; прямого доступу до буфера у нього немає. Спочатку процедура перевіряє умову *notfull*, щоб з'ясувати, чи є в буфері порожнє місце. Якщо його немає, процес призупиняється, і до монітора може увійти інший процес (виробник або споживач). Пізніше, коли в буфері з'явиться вільне місце, призупинений процес витягається з черги і поновлює свою роботу. Після того, як процес помістить символ у буфер, він сигналізує про виконання умови *notempty*, що розблоковує процес споживача, якщо він був призупинений.

Цей приклад ілюструє розділення відповідальності при роботі з монітором і при використанні семафора. **Монітор автоматично забезпечує взаємовиключення**: одночасне звернення виробника і споживача до буфера неможливе.

Проте програміст повинен коректно розмістити всередині монітора примітиви *Cwait* і *Csignal* для того, щоб запобігти розміщенню елемента в заповненому буфері, або вибірці з порожнього буфера. У разі використання семафорів відповідальність як за синхронізацію, так і за взаємовиключення повністю лежить на програмістові.

Слід звернути увагу, що в лістингу 8.7 процес покидає монітор негайно після виконання функції *Csignal.* Якщо виклик *Csignal* здійснюється не в кінці процедури, то, за пропозицією Хоара, процес, що викликав цю функцію, призупиняється для того, щоб звільнити монітор для іншого процесу,

поміщається в чергу призупинених процесів і залишається там до тих пір, поки монітор знову не звільнитися. Якщо виконання умови *x* не чекає жоден процес, виклик *Csignal(x)* не виконує ніяких дій.

Як при роботі з семафорами, так і при роботі з моніторами легко допустити помилку у функції синхронізації. Наприклад, якщо опустити будь-який з викликів *Csignal* у моніторі, то процес, що потрапив у відповідну чергу, залишиться там назавжди. Перевага моніторів порівняно з семафорами в тому, що всі синхронізуючі функції знаходяться в моніторі. Таким чином, перевірити коректність синхронізації і відловити можливі помилки виявляється простіше, ніж при використанні семафорів. Крім того, при правильно розробленому моніторі доступ до захищених ресурсів коректний незалежно від запитуваного процесу. При використанні ж семафорів доступ до ресурсу коректний тільки в тому випадку, якщо правильно розроблені всі процеси, що звертаються до ресурсу.

8.2.2 Монітори із сповіщенням і широкомовленням

Визначення монітора, дане Хоаром, вимагає, щоб у разі, якщо черга очікування виконання умови не порожня, то при виконанні яким-небудь процесом операції *Csignal* для цієї умови був негайно запущений процес, що знаходиться у вказаній черзі. Таким чином, процес, що виконує операцію *Csignal,* повинен або негайно вийти з монітора, або бути призупиненим. У такого підходу є два недоліки:

1. Якщо, виконуючий операцію *Csignal*, процес не завершив своє перебування, то потрібно два додаткових перемикання процесів: один для призупинення цього процесу, а другий для відновлення його роботи, коли монітор стане доступний.
2. Планувальник процесів, пов'язаний з сигналом, має бути ідеально надійний. При виконанні *Csignal* процес з відповідної черги має бути негайно активізований, причому планувальник повинен гарантувати, що до активізації ніякий інший процес не увійде до монітора. Інакше умова, з якою активізується процес, може встигнути змінитися. Так, наприклад, в лістингу 8.7, коли виконується *Csignal(notempty),* процес з черги *notempty* має бути активізований до того, як новий споживач увійде до монітора. Якщо станеться збій процесу виробника безпосередньо після того, як він додасть символ, так що операція *Csignal* не буде виконана, то в результаті процеси в черзі *notempty* виявляться навіки заблоковані.

Лемпсон (Lampson) і **Ределл** (Redell) розробили інше визначення монітора для мови програмування *Mesa*, Їх підхід дозволяє долати описані проблеми, а крім того, надає ряд корисних розширень концепції моніторів. Структура монітора **Mesa** використана і в мові програмування Modula-3 [12].

У мові програмування *Mesa* примітив *Csignal* був замінений примітивом *Cnotify,* який інтерпретується таким чином. Коли процес, що виконується в моніторі, викликає *Cnotify(x),* про це повідомляється черга умови *x*, але виконання процесу, що викликав *Cnotify*, триває. Результат повідомлення

полягає в тому, що процес на початку черги умови відновить свою роботу в найближчому майбутньому, коли монітор виявиться вільним. Проте оскільки немає гарантії, що деякий інший процес не ввійде до монітора до згаданого очікуючого процесу, при відновленні роботи наш процес повинен ще раз перевірити, чи виконана умова. У разі використання такого підходу процедури монітора *ProducerConsumer* матимуть такий вигляд (лістинг 8.8).

Лістинг 8.8. Код монітора ProducerConsumer

```
Void Append(char x) {
	While(count == N)
		cwait(notfull);	/* Буфер заповнений */
	Buffer [nextin] = x;
	Nextin % N;
	Count++;  /* Додаємо елемент у буфер */
	Cnotify(notempty); /* Повідомляємо  споживача */
}
Void Take(char x) {
	While(count == 0)
		cwait(notempty);/* Буфер порожній */
	x = Buffer [nextout];
	Nextout % N;
	Count--;	/* Видаляємо елемент з буферу */
	Cnotify(notfull);	/* Повідомляємо виробника */
}
```

Інструкції *if* замінені циклами *while.* Таким чином, виконуватиметься як мінімум одне зайве обчислення змінної умови. Проте в цьому випадку відсутні зайві перемикання процесів, і немає обмежень на момент запуску очікуючого процесу після виклику *Cnotify*.

Однією з корисних особливостей такого роду моніторів може бути пов'язаний з кожним примітивом умови *Cnotify* граничний час очікування. Процес, який прочекав повідомлення впродовж граничного часу, поміщається в список активних незалежно від того, було повідомлення про виконання умови чи ні. Така можливість запобігає нескінченному голодуванню процесу в разі збою інших процесів перед повідомленням про виконання умови.

При використанні правила, згідно з яким відбувається повідомлення процесу, а не його насильницька активація, в систему команд можна включити примітив (наприклад, *cbroadcast* – передача), який викликає активацію всіх очікуючих процесів. Це може бути в ситуаціях, коли процес не обізнаний про кількість очікуючих процесів.

Припустимо, наприклад, що в програмі «виробник-споживач» функції *Append* і *Take* можуть працювати з символьними блоками змінної довжини. У цьому випадку, коли виробник додає в буфер блок символів, він не зобов'язаний знати, скільки символів готовий спожити кожен з очікуючих процесів. Він просто виконує інструкцію *cbroadcast*, і усі очікуючі процеси отримають повідомлення про те, що вони можуть спробувати отримати свою частку символів з буфера (широкомовне повідомлення).

Крім того, широкомовне повідомлення може використовуватися в тому випадку, коли процес не в змозі точно визначити, який саме процес з очікуючих має бути активований. Хорошим прикладом такої ситуації може служити диспетчер пам'яті. Припустимо, що у нас є *j* байт вільної пам'яті, і деякий процес звільняє додатково *k* байт. Диспетчерові не відомо, який саме з очікуючих процесів зможе працювати з *j*+*k* байт вільної пам'яті; отже, він може використати виклик *cbroadcast*, і всі очікуючі процеси самі перевірять, чи достатньо їм вільної пам'яті.

Перевага монітора Лемпсона-Ределла порівняно з монітором Хоара є його менша схильність до помилок. Оскільки кожна процедура після отримання сигналу перевіряє змінну монітора з використанням циклу *While*, то навіть при передачі процесом невірного або широкомовного повідомлення – це не призведе до помилки в програмі, що отримала сигнал. Просто переконавшись, що її даремно активізували, програма знову перейде в стан очікування.

Наведемо приклад програми «Виробник-Споживач» на мові програмування Java (лістинг 8.9). Java – об'єктно-орієнтована мова програмування, що підтримує потоки на рівні користувача, і що дозволяє групувати методи (процедури) в класи. Додавання в опис методу ключового слова *synchronized* гарантує, що якщо хоч би один потік почав виконання цього методу, жоден інший потік не зможе виконати інший синхронізований (визначений як *synchronized*) метод з цього класу.

Лістинг 8.9 Розв'язок задачі «Виробник-Споживач» на Java [12]

```
public class ProducerConsumer {
	static final int N = 100;	// Розмір буфера
	//створити екземпляр потоку виробника
	static producer p = new producer();
	// створити екземпляр потоку споживача
	static consumer c = new consumer();
	//створити екземпляр монітора
	static our_monitor mon = new our_monitor();
	public static void main(String args[]) {
		p.start();	// запуск потоку виробника
		c.start();	// запуск потоку споживача
	}
	static class producer extends Thread {
	  public void run(){	// метод run містить програму потоку
		int item;
		while(true) {	// цикл виробника
			item  = produce_item();
			mon.insert(item);
		}
	  }
	  private int produce_item(){.} // виробництво
	}
	static class consumer extends Thread {
	  public void run(){ // метод run містить програму потоку
		int item;
```

```
            while(true) {       // цикл споживача
                    item  = mon.remove();
                    consume_item(item);
            }
        }
        private void consume_item(int item) }
            // споживання
        }
      static class our_monitor {      // монітор
        private int buffer[] = new int[N];
        private int count = 0, lo = 0, hi = 0;     // лічильник і індекси
        public synchronized void insert(int val) {
        if(count == N) go_to_sleep();      //якщо буфер повний – чекати
        buffer[hi] = val;               // помістити елемент у буфер
        hi % N; // наступний сегмент для елементу
        count = count+1; // лічильник елементів у буфері
        if(count == 1) notify(); // якщо споживач в стані
      }                            // очікування, то активувати його
      public synchronized int remove() {
        int val;
        if(count == 0) go_to_sleep();     //якщо буфер порожній – чекати
        val = buffer[lo] ; // забрати елемент з буфера
        lo % N; // наступний сегмент для витягання
        count = count - 1; //тепер у буфері на 1 елемент менше
        if(count == N - 1) notify(); //якщо виробник в стані
                    // очікування, то активувати його
        return val;
      }
      private void go_to_sleep() {
            try { wait(); }
            catch(interruptedException exc);
      }
      }
}
```

У програмі виробника є нескінченний цикл формування даних і розміщення їх у загальному буфері. У коді споживача є нескінченний цикл з вилученням даних із загального буфера. Клас *our_monitor* містить буфер, змінні адміністрування і два методи синхронізації. Коли виробник активний в процедурі *insert*, споживач не може бути активний в процедурі *remove*, що виключає стан змагання. Змінна *cont* містить кількість елементів у буфері, набуваючи значень від 0 до *N-1*. Змінна *lo* є індексом наступного буфера, з якого слід витягнути дані. Змінна *hi* є індексом наступного буфера, в який слід помістити дані. Розв'зана ситуація, коли *lo* = *hi*, що означає 0 або *N* елементів у буфері. Відрізняти ці два випадки можна за змінною *count*.

Синхронізовані методи в мові Java відрізняються від стандартних моніторів відсутністю змінних стану. Натомість пропонується дві процедури *wait* і *notify*, які використовуються в синхронізованих методах. Процедура може бути перервана, для чого і служить увесь набір програм, що оточує її. У нашому

випадку *go_to_sleep* описує відхід в стан очікування. Останній приклад написаний на Java, а не на *C*, як усі інші приклади, оскільки *C* і багато інших мов не мають моніторів (окрім Modula-3).

8.3 Передача повідомлень

Семафори і монітори взагалі-то були розроблені для розв'язання задач взаємного виключення в системі з одним або декількома процесорами, що мають доступ до загальної пам'яті. Ці примітиви непридатні в розподіленій системі, що складається з декількох процесорів з власною пам'яттю у кожного, оскільки всі вони базуються на використанні оперативної пам'яті, що розділяється.

Наприклад, якщо два процеси виконуються на одній і тій же машині, вони можуть мати спільний доступ до семафора, який зберігається, наприклад, в ядрі. Проте, якщо процеси виконуються на різних машинах, то цей метод не застосовний, для розподілених систем потрібні інші підходи.

Висновок з усього вищесказаного такий: семафори є примітивами занадто низького рівня, а монітори можуть використовуватися тільки в деяких мовах програмування. Ці ж примітиви не підходять для реалізації обміну інформацією між комп'ютерами – потрібне щось інше.

8.3.1 Примітиви передачі повідомлень

При взаємодії процесів між собою повинні задовольнятися дві фундаментальні вимоги: синхронізації і комунікації. Процеси мають бути синхронізовані, щоб забезпечити виконання взаємних виключень. Співпрацюючі процеси повинні мати можливість обмінюватися інформацією. Одним з підходів до забезпечення обох вказаних функцій є ***передача повідомлень.*** Важливою перевагою передачі повідомлень є її придатність для реалізації як в одно- так і багатопроцесорних системах з пам'яттю, що розділяється, так і в розподілених системах. Для зберігання відправленного, але ще не отриманого повідомлення потрібне місце. Воно називається ***буфером повідомлень***, або ***поштовою скринькою***.

З системами передачі повідомлень пов'язана велика кількість складних проблем і конструктивних питань, які не виникають у разі семафорів і моніторів. Особливо багато складнощів з'являється в разі взаємодії процесів, що відбуваються на різних комп'ютерах, сполучених мережею. Так, повідомлення може загубитися в мережі. При розробці систем передачі повідомлень слід розв'язати низку запитань, які перераховані на рис. 8.3.

Системи передачі повідомлень можуть бути різних типів, ми ж розглянемо найзагальніші можливості і властивості таких систем. Функції передачі повідомлень представлені у вигляді пари примітивів

send*(<одержувач >, <повідомлення>)* і

receive*(<відправник >,< повідомлення >),*

які є швидше системними викликами, ніж структурними компонентами мови, що відрізняє їх від моніторів і робить схожими на семафори.

Синхронізація Відправлення Блокуюче Неблокуюче Отримання Блокуюче Неблокуюче Перевірка наявності	**Формат** Вміст Довжина Фіксована Мінлива
Адресація Пряма Відправлення Отримання Неявне Явне Непряма Статична Динамічна Володіння	**Принцип роботи черги** FIFO Пріоритетна

Рисунок 8.3 – Характеристики систем передачі повідомлень

Процес відправляє інформацію у виді <*повідомлення*> іншому процесу, визначеному як <*одержувач*>, викликом *send.* Отримує інформацію процес за допомогою виконання примітиву *receive,* якому вказує відправник повідомлення.

8.3.2 Синхронізація

Розглянемо, що відбувається після того, як процес викликає примітиви *send* або *receive.* При виконанні *send* є дві можливості: або процес, що відправляє повідомлення, блокується, або продовжує роботу. Аналогічно є дві можливості і в процесу, що виконує примітив *receive*.

1. Якщо повідомлення було заздалегідь відправлене, то процес отримує його і продовжує роботу.
2. Якщо повідомлення, яке очікує отримання, немає, то:
   a) або процес блокується, поки повідомлення не буде отримано;
   b) або процес продовжує виконання, відмовляючись від подальших спроб отримати повідомлення.

Таким чином, і відправник, і одержувач можуть бути такими, що блокуються або не блокуються. Зазвичай зустрічається три комбінації, наведені нижче.

1. **Блокуюче відправлення, блокуюче отримання.** І відправник, і одержувач блокуються до тих пір, поки повідомлення не буде доставлено за призначенням. Таку ситуацію іноді називають ***рандеву***. Ця комбінація забезпечує тісну синхронізацію процесу.

2. **Неблокуюче відправлення, блокуюче отримання**. Хоча відправник і може продовжувати роботу, одержувач блокується до отримання повідомлення. Ця комбінація зустрічається найчастіше. Вона дозволяє процесу відправляти одне або декілька повідомлень різним одержувачам з максимальною швидкістю. Прикладом може служити серверний процес, існуючий для надання сервісів або ресурсів іншим процесам.
3. **Неблокуюче відправлення, неблокуюче отримання**.

Неблокуючий примітив *send* найприродніший для більшості задач з використанням паралельних обчислень. Але при такому підході на програміста покладається задача відстежування успішної доставки повідомлення адресатові. Процес-одержувач повинен, у свою чергу, відправити відповідь з підтвердженням отримання повідомлення. Інакше можлива ситуація, коли деяка помилка призведе до безперервної генерації повідомлень.

У разі використання примітиву *receive* для більшості задач природною видається блокуюча технологія, оскільки, найчастіше, процес, що запросив інформацію, потребує її для продовження роботи. Правда, якщо повідомлення втрачається, одержувач може виявитися назавжди заблокованим. Розв'язати цю проблему можна за допомогою неблокованого примітиву *receive*. Проте у цього варіанту є своє слабке місце: якщо повідомлення відправлене після того, як процес виконав операцію *receive*, то воно виявляється втраченим. Ще один можливий підхід для вирішення цієї проблеми полягає в тому, що перед тим, як виконати *receive*, необхідно перевірити, чи немає повідомлення, що очікує отримання.

8.3.3 Адресація

Різні схеми визначення процесів у примітивах *send* і *receive* розділяються на дві категорії: пряму і непряму адресацію.

При **прямій адресації** примітив *send* включає ідентифікатор процесу-одержувача. Коли застосовується примітив *receive*, можна піти двома шляхами. Перший шлях полягає у вимозі явної вказівки процесу-відправнику, тобто процес повинен знати заздалегідь, від якого саме процесу він чекає повідомлення. Такий шлях досить ефективний, якщо паралельні процеси співпрацюють.

Проте в багатьох випадках неможливо передбачити, який процес буде відправником очікуваного повідомлення. Як приклад можна навести процес сервера друку, який приймає повідомлення – запити на друк від будь-якого іншого процесу. Для таких додатків ефективнішим буде підхід з використанням неявної адресації. У цьому випадку параметр <*відправник*> набуває значення, яке повертається після виконання операції отримання повідомлення.

Ще одним поширеним підходом є **непряма адресація.** Вона припускає, що повідомлення відправляються не прямо від відправника одержувачеві, а направляється в спільно використовувану структуру даних, що складається з черг для тимчасового зберігання повідомлень. Такі черги іменують ***поштовими скриньками*** (mailbox). Таким чином, для зв'язку між двома процесами один з них

посилає повідомлення у відповідну поштову скриньку, з якої його забирає другий процес.

Ефективність непрямої адресації полягає в гнучкості використання повідомлень. При такій схемі роботи з повідомленнями відношення між відправником і одержувачем можуть бути будь-якими – «один до одного», «один до багатьох», «багато до одного» або «багато до багатьох».

Відношення «один до одного» забезпечує закритий зв'язок між двома процесами, ізолюючи їх від стороннього втручання. Відношення «багато до одного» корисно при взаємодії клієнт/сервер – один процес при цьому є сервером, обслуговуючим багато клієнтів. У такому разі про поштову скриньку говорять як про ***порт***. Відношення «один до багатьох» забезпечує розсилку від одного процесу багатьом одержувачам, дозволяючи здійснити широкомовне повідомлення процесам.

Зв'язок процесів з поштовими скриньками може бути як статичним, так і динамічним. Порти найчастіше статистично пов'язані з певними процесами. Тобто, порт створюється і назначається процесу назавжди. Те ж спостерігається і у разі використання відношення «один до одного» – закриті канали зв'язку, як правило, також визначаються статично, раз і назавжди.

За наявності декількох відправників їх зв'язок з поштовою скринькою може здійснюватися динамічно з використанням для цієї мети додаткових примітивів *connect* і *disconnect*.

8.3.4 Формат повідомлення

Формат повідомлення залежить від переслідуваних цілей і від того, чи працює система передачі повідомлень на одному комп'ютері або в розподіленій системі. У ряді ОС розробники віддають перевагу коротким повідомленням фіксованої довжини, що дозволяє мінімізувати обробку і зменшити витрати пам'яті на їх зберігання. Проте гнучкіший підхід дозволяє використати повідомлення змінної довжини. На рис. 8.4 показаний формат типового повідомлення змінної довжини.

Тип повідомлення
Ідентифікатор одержувача
Ідентифікатор відправника
Довжина повідомлення
Управляюча інформація
Зміст повідомлення

Рисунок 8.4 – Узагальнений формат повідомлення

8.3.5. Взаємні виключення

У лістингу 8.10 показаний один із способів реалізації взаємних виключень з використанням системи передачі повідомлень. У цій програмі передбачається використання блокуючого *receive* і неблокуючого *send.* Певна кількість процесів,

що паралельно виконуються, спільно використовують поштову скриньку *mutex* як для відправки повідомлень, так і для їх отримання. Поштова скринька після ініціалізації містить єдине повідомлення з порожнім змістом. Процес, що має намір увійти до критичного розділу, спочатку намагається отримати повідомлення. Якщо поштова скринька порожня, процес блокується. Як тільки процес отримує повідомлення, він тут же виконує критичний розділ і потім посилає повідомлення назад в поштову скриньку.

Лістинг 8.10. Реалізація взаємних виключень з використанням повідомлень

```
const int n = /* Кількість процесів */;
void P(int n)  {
       message  msg;
       while(true)  {
              receive(mutex, msg);
              /* Критичний розділ */;
              send(mutex, msg);
              /* Інший код */;
       }
}
void main()  {
       create_mailbox(mutex);
       send(mutex, null);
       parbrgin(P(1),P(2),…,p(n));
}
```

У розглянутому рішенні передбачається, що якщо операція *receive* виконується паралельно більш ніж одним процесом, то:

- якщо є повідомлення, воно передається тільки одному з процесів, а інші процеси блокуються;
- якщо черга повідомлень порожня, блокуються усі процеси; при появі в черзі повідомлення, його отримує тільки один із заблокованих процесів.

Це припущення виконується практично для всіх засобів передачі повідомлень.

В якості іншого прикладу використання повідомлень в лістингу 8.11 наведено рішення задачі Виробник-Споживач з обмеженим буфером.

Лістинг 8.11. Рішення задачі Виробник-Споживач з обмеженим буфером і з використанням повідомлень

```
const int capacity = /* Емкость буфера */;
Null   = /* Порожнє повідомлення */;
int I;
void producer()  {
       message pmsg;
       while(true)  {
              receive(myproduce, pmsg);
              pmsg = produce();
              send(mayconsume, pmsg);
       }
```

```
}
void consumer() {
      message cmsg;
      while(true) {
            receive(myconsume, cmsg);
            consume(cmsg);
            send(mayproduce, null);
      }
}
void main() {
      create_mailbox(mayproduce);
      create_mailbox(mayconsume);
      for (int I = 1; I <= capacity; I++) send(mayproduce, null);
      parbegin(producer, consumer);
}
```

На відміну від семафорів тут передаються повідомлення, а не сигнали. У програмі використовуються дві поштові скриньки. Коли виробник генерує дані, він посилає їх в якості повідомлення в поштову скриньку *mayconsume*. Поки в цій поштовій скриньці є хоч би одне повідомлення, споживач може отримати дані. Отже, поштова скринька *mayconsume* служить буфером, дані в якому організовані у вигляді черги повідомлень. «Місткість» цього буфера визначається глобальною змінною *capacity*. Поштова скринька *mayproduce* спочатку заповнена порожніми повідомленнями в кількості, рівній місткості буфера. Кількість повідомлень в цій поштовій скриньці зменшується при кожному прибутті нових даних і збільшується при їх використанні.

Такий підхід досить гнучкий – він може працювати з будь-якою кількістю виробників і споживачів. Більш того, система виробників і споживачів може бути розподіленою, коли усі виробники і поштова скринька *mayproduce* знаходяться на одній машині, а споживачі і поштова скринька *mayconsume* – на іншій.

Контрольні питання і тести до розділу 8

Контрольні питання

1. Який узагальнений засіб запропонував Дейкстра для синхронізації процесів, ввівши два нові примітиви?
2. Які дві операції визначені над семафором?
3. Як називається спрощена версія семафора?
4. Який семафор називається сильним і слабким семафором?
5. Дайте визначення узагальненого семафора?
6. Ким уперше було дано формальне визначення концепції монітора?
7. Який монітор розробили Лемпсон (Lampson) і Ределл (Redell)?
8. Перевага монітора Лемпсона-Ределла в порівнянні з монітором Хоара.
9. Які засоби застосовуються для синхронізації процесів, якщо процеси виконуються на різних машинах?
10. Чи можуть і відправник, і одержувач повідомлень бути такими, що блокуються або не блокуються?
11. Які черги іменують «поштовими скриньками»

12. У чому відмінність прямої і непрямої адресація повідомлень?
13. Чи правда, що в будь-який момент часу потік може знаходиться в черзі очікування тільки одного семафора?
14. Що станеться, якщо потік викличе операцію семафора *V*, не викликавши перед цим операцію *P*?
15. Що станеться, якщо при заблокованих семафором потоків не дотримуватиметься дисципліна «Перший прийшов – Перший пішов»?

Тести

1. Для реалізації синхронізації на рівні мови програмування використовуються високорівневі примітиви:
   1) супервізори;
   2) монітори;
   3) семафори;
   4) маркери.
2. Нехай є два паралельні процеси. В одному виконується код P(S1); P(S2), в іншому – код P(S2); P(S1), де S1 і S2 – семафори. Як поводитиметься програма?
   1) станеться взаємне блокування процесів;
   2) перший процес заблокується;
   3) другий процес заблокується;
   4) обидва процеси автоматично завершать своє виконання.
3. Скільки символів **D** буде надруковано після запуску трьох потоків, які синхронізуються за допомогою семафорів:

semsphore L=3, R=0;
void thread_1( ) {
  for(; ;) {
   down(L);
   printf(‘C’);
   up(R);
  }
}
	void thread_2( ) {
	  for(I=1; I=5; I++) {
	   down(R);
	   printf(‘A’);
	   printf(‘B’);
	   up(R);
	  }
	}
		void thread_3( ) {
		  for(; ;) {
		   down(R);
		   printf(‘D’);
		  }
		}

   1) 1;
   2) 2;
   3) 3;
   4) 4.
4. Яким чином монітор запобігає одночасному виконанню декількох потоків усередині монітора?
   1) взаємовиключення гарантується монітором за допомогою спеціальної мікросхеми «Монітор»;
   2) взаємовиключення гарантується монітором, оскільки монітор – це спеціальне облаштування типу процесора, в якому потік виконує свою критичну секцію за певний квант часу як неділима (атомарна) операція;

3) взаємовиключення гарантується всередині монітором за допомогою програмних алгоритмів, або апаратних засобів, або семафорів, розглянутих раніше до моніторів;
4) взаємовиключення гарантується монітором, оскільки в цьому пристрої виконується мікрокоманда, яку може виконувати в певний момент часу тільки один потік, так само як в певний момент часу в процесорі може виконуватися тільки один потік (процес).

5. Чому потік повинен чекати звільнення ресурсу поза монітором?
   1) тому що очікування звільнення ресурсу потоком поза монітором гарантує його отримання в будь-який момент часу, чим очікування його всередині монітора;
   2) якби потік чекав звільнення ресурсу всередині монітора, то жоден інший потік не зміг би увійти до монітора, щоб повернути ресурс системі;
   3) тому що очікування звільнення ресурсу потоком поза монітором гарантує його роботу для отримання цього ресурсу у іншого монітора;
   4) якби потік чекав звільнення ресурсу всередині монітора, то всі інші потоки також змогли б увійти до монітора і конкурувати за цей ресурс.
6. 6. М'ютекс – це синхронізуючий об'єкт, що є окремим випадком:
   1) монітора Хоара;
   2) події;
   3) об'єкта критичної секції;
   4) семафора.
7. У якій області пам'яті зберігається змінна-семафор:
   1) у загальній області пам'яті процесів, синхронізація яких здійснюється за допомогою семафора;
   2) у адресному просторі ядра операційної системи;
   3) у спеціальному регістрі процесора;
   4) у файлі семафора, який використовується усіма синхронізуючими процесами.
8. Чи вірно, що в будь-який момент часу потік може знаходитися в черзі очікування тільки одного семафора?
   1) так;
   2) ні.
9. Чи вірно, що кожен монітор має рівно одну змінну-умову?
   - так;
   - ні.

9 ВЗАЄМНЕ БЛОКУВАННЯ

У мультипрограмній системі процес знаходиться в стані *тупика, дедлока,* або *клінчу,* якщо він чекає деякої події, яка ніколи не станеться. Системна тупикова ситуація, або «зависання» системи – це ситуація, коли один або більше процесів виявляються в стані тупика. У мультипрограмних обчислювальних машинах однією з головних функцій операційної системи є розподіл ресурсів. Коли ресурси розподіляються між багатьма користувачами, кожному з яких надається право виняткового управління виділеними йому конкретними ресурсами, цілком можливо виникнення тупиків, із-за яких процеси деяких користувачів ніколи не зможуть завершитися.

У цьому розділі обговорюється проблема тупиків і наводяться основні результати наукових досліджень, присвячених питанням запобігання, обходу, виявленню тупиків і питанням відновлення після них. Розглядається також тісно пов'язана проблема нескінченного відкладання, коли процес, що навіть не знаходиться в стані тупика, чекає події, яка може ніколи не статися із-за «необ'єктивних» принципів, закладених в системі планування ресурсів.

Розглядаються можливі компромісні рішення з точки зору накладних витрат на включення засобів боротьби з тупиками і очікуваних від цього переваг. У деяких випадках ціна, яку доводиться платити за те, щоб зробити систему вільною від тупиків, занадто висока. В інших випадках, наприклад, в системах управління процесами реального часу, просто немає іншого вибору, оскільки виникнення тупиків може призвести до катастрофічних наслідків.

9.1 Принципи взаємного блокування

У комп'ютерних системах існує велика кількість ресурсів, кожен з яких в конкретний момент часу може використовуватися тільки одним процесом (неподільний ресурс). Поява двох процесів, які одночасно передають дані, наприклад, на принтер, призведе до друку безглуздого набору символів. Наявність двох процесів, що використовують один і той же елемент таблиці файлової системи, стане причиною руйнування файлової системи. Тому усі ОС мають здатність надавати процесу ексклюзивний доступ не до одного, а до декількох ресурсів.

У попередньому розділі були розглянуті способи синхронізації процесів, які дозволяють процесам успішно кооперуватися. Проте, якщо засобами синхронізації користуватися необережно, то можуть виникнути непередбачені утруднення – ***взаємні блокування***, що називаються також ***клінчами (clinch), дедлоками (deadlocks***) або ***тупиками***.

Взаємному блокуванню можна дати таке формальне визначення:

Взаємне блокування в групі процесів виникає в тому випадку, якщо кожен процес з цієї групи чекає події, настання якої залежить виключно від іншого процесу з цієї ж групи.

Розглянемо приклад тупика. Нехай двом потокам (процесам), що виконуються в режимі мультипрограмування, для виконання їх роботи потрібно два ресурси, наприклад, порт і диск. На рис. 9.1, *а* показані фрагменти

відповідних програм. І нехай після того, як потік *A* зайняв порт (встановив блокуючу змінну), він був перерваний. Управління отримав потік *B,* який спочатку зайняв диск, але при виконанні наступної команди був заблокований, оскільки порт виявився вже зайнятим потоком *A.* Управління знову отримав потік *A,* який відповідно до своєї програми зробив спробу зайняти диск і був заблокований: диск вже розподілений потоку *B.* У такому положенні потоки *A* і *B* можуть знаходитися скільки завгодно довго.

Залежно від співвідношення швидкостей потоків, вони можуть або абсолютно незалежно використати ресурси (*г*), що розділяються, або утворювати черги до ресурсів *(в),* що розділяються, або взаємно блокувати один одного *(б).*

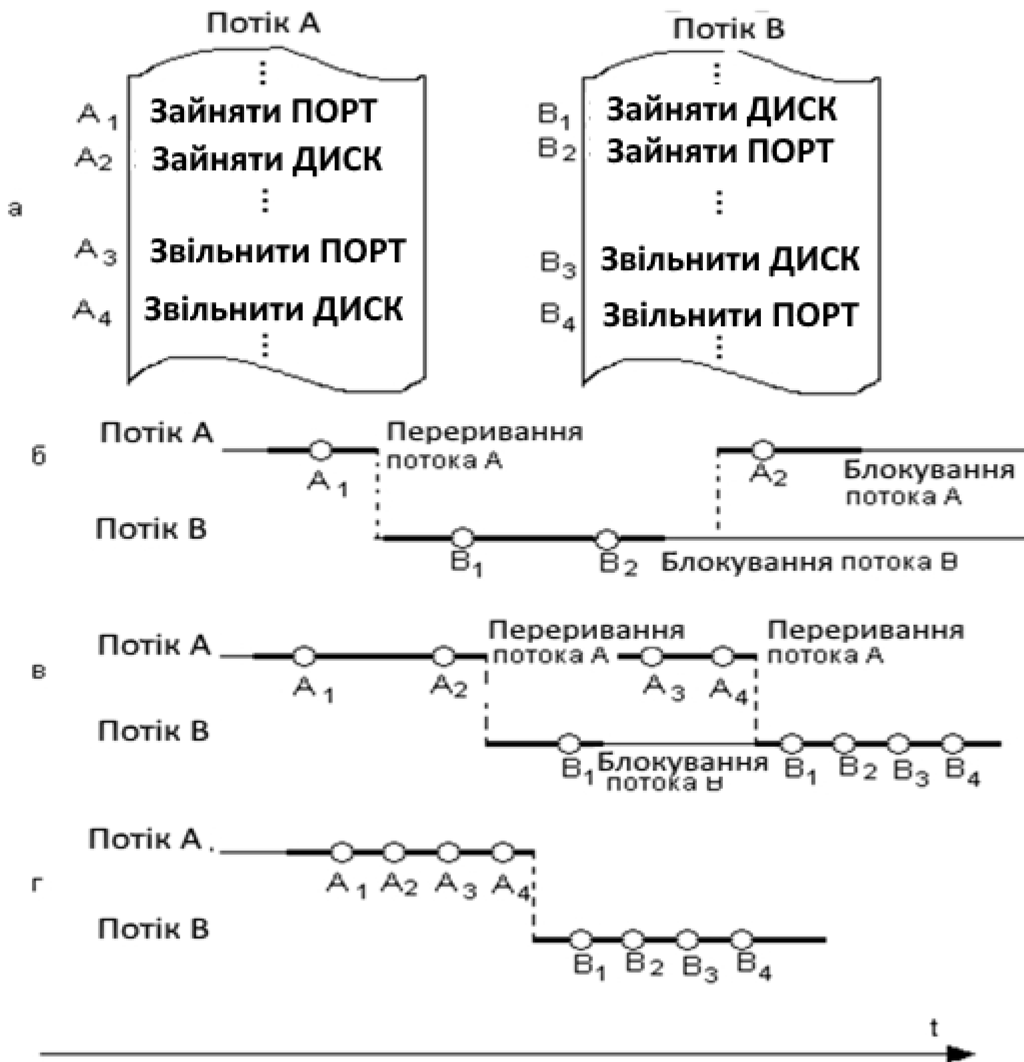


Рисунок 9.1 – Виникнення взаємних блокувань при виконанні програми; *(б)* – взаємне блокування (клінч); *(в)* – черга до диска, що розділяється; *(г)* – незалежне використання ресурсів

У розглянутих прикладах тупик був утворений двома потоками, але взаємно блокувати один одного можуть і більше число потоків.

Тупикам можна запобігти на стадії написання програм, тобто програми мають бути написані так, щоб тупики не могла виникнути ні при якому співвідношенні взаємних швидкостей потоків. Так, якби в попередньому

прикладі потік *A* і потік *B* просили ресурси в однаковій послідовності, то тупик був би в принципі неможливий. Другий підхід до запобігання тупиків називається динамічним і полягає у використанні певних правил при призначенні ресурсів процесам, наприклад, ресурси можуть виділятися в певній послідовності, загальній для усіх процесів.

Існують формальні, програмно-реалізовані методи розпізнавання тупиків, засновані на веденні таблиць розподілу ресурсів і таблиць запитів до зайнятих ресурсів. Аналіз цих таблиць дозволяє виявити взаємні блокування.

Взаємні блокування можуть статися в більшості інших ситуацій окрім запитів виділених пристроїв введення-виведення. У системах баз даних програма може виявитися вимушеною заблокувати декілька записів, щоб уникнути стану конкуренції. Якщо процес *A* блокує запис *R1,* процес *B* блокує запис *R2,* а потім кожен процес намагається заблокувати чужий запис, ми також опинимося в тупику. Таким чином, взаємні блокування з'являються при роботі як з апаратними, так і з програмними ресурсами.

Як ми вже знаємо, ресурси бувають двох видів: вивантажувані і невивантажувані.

До **вивантажуваних** належать такі ресурси, які можуть бути безболісно відібрані в процесу, який їх має. Прикладом такого ресурсу може послужити пам'ять. Розглянемо систему, що має 256 Мб призначеної для користувача пам'яті, один принтер і два процеси по 256 Мб, кожен з яких хоче щось вивести на друк. Процес *A* просить і отримує принтер, а потім починає обчислювати значення, призначене для виведення на друк. Але до завершення обчислення закінчується виділений йому квант часу, і він вивантажується на диск.

Тепер запускається процес *B,* який безуспішно намагається оволодіти принтером. Потенційно виникає ситуація взаємного блокування, оскільки у процесу *A* є принтер, а у процесу *B* – пам'ять, і жоден з них не може продовжити свою роботу без ресурсу, що утримується іншим процесом.

На щастя, є можливість відібрати пам'ять у процесу *B,* вивантаживши цей процес на диск, і завантажити звідти процес *A.* Тепер *A* може відновити свою роботу, виконати друк і вивільнити принтер. І ніякого взаємного блокування не виникне.

А ось **невивантажуваний** ресурс не можна відібрати у його поточного власника, не викликавши збою в обчисленнях. Якщо у процесу, який вже приступив до запису на компакт-диск, несподівано відібрати пишущий привід і віддати його іншому процесу, то це призведе до псування компакт-диска. Тобто, приводи компакт-дисків не можна відібрати в довільний момент. Як правило, у взаємних блокуваннях фігурують невивантажувані ресурси. Взаємні блокування за участю вивантажуваних ресурсів можуть бути усунені шляхом перерозподілу ресурсів від одного процесу до іншого. Тому наша увага буде сконцентрована на невивантажуваних ресурсах.

Проблема тупиків включає такі задачі:

- запобігання тупиків;
- розпізнавання тупиків;
- відновлення системи після тупиків.

У деяких випадках, коли тупикова ситуація утворена багатьма процесами, що використовують багато ресурсів, розпізнавання тупиків є нетривіальною задачею.

Якщо ж тупикова ситуація виникла, то не обов'язково знімати з виконання усі заблоковані процеси. Можна зняти тільки частину з них, при цьому звільняються ресурси, очікувані іншими процесами. Можна повернути деякі процеси в область свопінгу, можна вчинити «відкат» деяких процесів до так званої контрольної точки, в якій запам'ятовується уся інформація, необхідна для відновлення виконання програми з цього місця.

9.2 Умови виникнення тупиків

Для того щоб взаємне блокування стала можливим, повинні виконуватися чотири необхідні умови (Коффман, Елфік і Шошані, 1971 р.) [12].

1. **Умова взаємного виключення (**Mutual exclusion). Кожен ресурс або виділений в даний момент тільки одному процесу, або доступний.
2. **Умова утримання і очікування ресурсів**. Процес може утримувати виділені ресурси під час запиту (очікування) інших ресурсів.
3. **Умова невивантажуваності (відсутності перерозподілу)**. У процесу не можна примусово забрати раніше отримані ресурси. Процес, що володіє ними, повинен сам звільнити ресурси.
4. **Умова циклічного очікування**. Існує замкнутий ланцюг процесів, кожен з яких утримує як мінімум один ресурс, необхідний процесу, наступному в ланцюзі після даного (рис. 9.2).

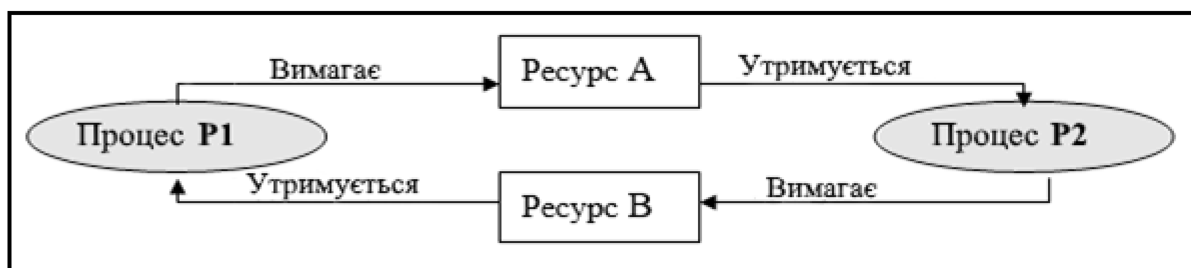


Рисунок 9.2 – Циклічне очікування ресурсів

Для того щоб сталася взаємне блокування, повинні виконуватися усі ці чотири умови. Якщо хоч би одна з них відсутня, тупикова ситуація неможлива.

9.3 Моделювання взаємного блокування

Холт (Holt, 1972 р.) показав, як ці чотири умови можуть бути змодельовані з використанням спрямованих графів. У графів є два види вузлів: процеси, показані колами, і ресурси, показані квадратами. Спрямоване ребро (дуга), яке слідує від вузла ресурсу (квадрата) до вузла процесу (кола), означає, що цей ресурс був раніше запрошений, отриманий і на даний момент утримується цим процесом [10].

Розглянемо, наприклад, два процеси – P1 і P2, і два ресурси – R1 і R2. Припустимо, що кожному процесу для виконання частини своїх функцій

потрібен доступ до загальних ресурсів. Тоді можливе виникнення такої ситуації: ОС виділяє ресурс R1 процесу P2, а ресурс R2 – процесу P1 (рис. 9.3, *а).* В результаті кожен процес чекає отримання одного з двох ресурсів. При цьому жоден з них не звільняє вже наявний ресурс, чекаючи отримання другого ресурсу для виконання функцій, що вимагають наявності двох ресурсів. У результаті процеси виявляються взаємно заблокованими як показано на рис. 9.3, *б,* або *в.*

У зв'язку з проблемою тупиків було виконано багато цікавих досліджень в області інформатики і операційних систем.

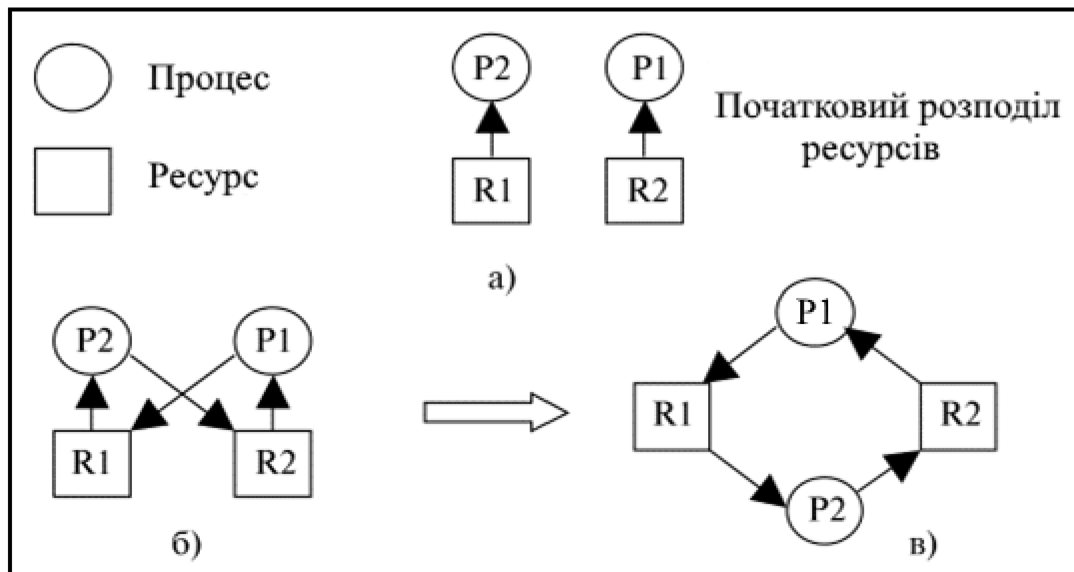


Рисунок 9.3 – Графи розподілу ресурсів

Основні напрямки боротьби з тупиками:

1. Ігнорувати цю проблему.
2. Виявлення і відновлення. Дайте взаємним блокуванням проявити себе, виявіть їх і зробіть необхідні дії.
3. Динамічне ухилення від них за рахунок ретельного розподілу ресурсів.
4. Запобігання за рахунок структурного придушення однієї з чотирьох необхідних умов для їх виникнення.

Стратегії запобігання взаємних блокувань дуже консервативні, оскільки вирішують проблему взаємних блокувань шляхом обмеження доступу процесів до ресурсів і накладення обмежень на процеси. Їх протилежність – стратегії виявлення і усунення взаємних блокувань, які не обмежують доступу процесів до ресурсів і не накладають ніяких обмежень на дії процесів.

Сучасні ОС періодично виконують алгоритми, за допомогою яких виявляється наявність взаємних блокувань у процесах. Перевірка наявності взаємних блокувань може виконуватися як при кожному запиті ресурсу, так і рідше, залежно від того, наскільки ймовірне виникнення взаємних блокувань. З одного боку, перевірка при кожному запиті має дві основні переваги: раніше виявлення і спрощення алгоритму. З іншого боку, така часта перевірка призводить до помітного споживання часу процесора. Узагальнені алгоритми виявлення і усунення взаємних блокувань можна знайти в літературі [9; 11].

9.4 Ігнорування проблеми взаємних блокувань

Простий підхід – не помічати проблему тупиків (***алгоритм страуса***). Для того щоб прийняти таке рішення, необхідно оцінити ймовірність виникнення взаємних блокувань і порівняти її з імовірністю збитку від інших відмов апаратного і програмного забезпечення. Проектувальники зазвичай не бажають жертвувати продуктивністю системи або зручністю користувачів для впровадження складних і дорогих засобів боротьби з тупиками.

Будь-яка ОС, що має в ядрі ряд масивів фіксованої розмірності, потенційно страждає від тупиків, навіть якщо вони не виявлені. Таблиця відкритих файлів, таблиця процесів, фактично кожна таблиця є обмеженим ресурсом. Заповнення усіх записів таблиці процесів може призвести до того, що черговий запит на створення процесу може бути відхилений. При несприятливому збігу обставин декілька процесів можуть видати такий запит одночасно і опинитися в тупику. Чи слід відмовлятися від виклику CreateProcess, щоб розв'язати цю проблему?

Підхід більшості популярних ОС (Unix, Windows та інших) полягає в тому, щоб ігнорувати цю проблему в припущенні, що малоймовірний випадковий тупик прийнятніший, ніж безглузді правила, що примушують користувачів обмежувати число процесів, відкритих файлів тощо.

9.5 Виявлення взаємних блокувань

При використанні технології виявлення і відновлення взаємних блокувань система не намагається уникнути взаємних блокувань. Вона дозволяє їм статися, намагається виявити момент їх виникнення, а потім робить деякі дії з відновлення працездатності. У цьому розділі будуть розглянуті деякі способи виявлення взаємних блокувань і деякі методи відновлення працездатності, які можна реалізувати.

9.5.1 Використання одного ресурсу кожного типу

Розпочнемо з найпростішого випадку, коли використовується тільки один ресурс кожного типу. У системи може бути один сканер, один привід компакт-дисків, один плотер і один накопичувач на магнітній стрічці, але тільки по одному екземпляру кожного класу ресурсів. Іншими словами, ми тимчасово виключаємо системи, які, наприклад, мають два принтери. Вони будуть розглянуті пізніше, з використанням іншого методу.

Для такої системи можна побудувати ресурсний граф, показаний на рис. 9.3. Якщо цей граф містить один і більше циклів, значить ми маємо справу зі взаємним блокуванням. Будь-який процес, що є частиною циклу, буде заблокований. Якщо циклів немає, значить, система не знаходиться в стані взаємного блокування.

Як приклад складнішої системи порівняно з раніше розглянутими, візьмемо систему з сімома процесами від *A* до *G,* і шістьма ресурсами від *R* до *W*. Кожен ресурс знаходиться в стані поточної зайнятості, і на кожен ресурс в даний момент поступив запит:

1. Процес A утримує R і хоче отримати S.
2. Процес B не утримує ніяких ресурсів, але хоче отримати T.
3. Процес З не утримує ніяких ресурсів, але хоче отримати S.
4. Процес D утримує U і хоче отримати S і T.
5. Процес E утримує T і хоче отримати V.
6. Процес F утримує W і хоче отримати S.
7. Процес G утримує V і хоче отримати U.

Виникає питання: «Чи знаходиться ця система в стані взаємного блокування, і якщо знаходиться, то які процеси залучені в цей стан»?. Щоб відповісти на це питання, можна побудувати граф ресурсів і процесів, показаний на рис. 9.4. Цей граф містить один цикл, який можна виявити візуально. Цей цикл показаний на рис. 9.4, *б*. З циклу видно, що процеси *D, E* і *G* залучені у взаємному блокуванні. Процеси *A, C* і *F* не знаходяться в стані взаємного блокування, оскільки ресурс *S* може бути виділений будь-якому з них, який потім закінчить свою роботу і поверне ресурс. Два процеси, які залишилися, зможуть узяти його по черзі і також завершити свою роботу.

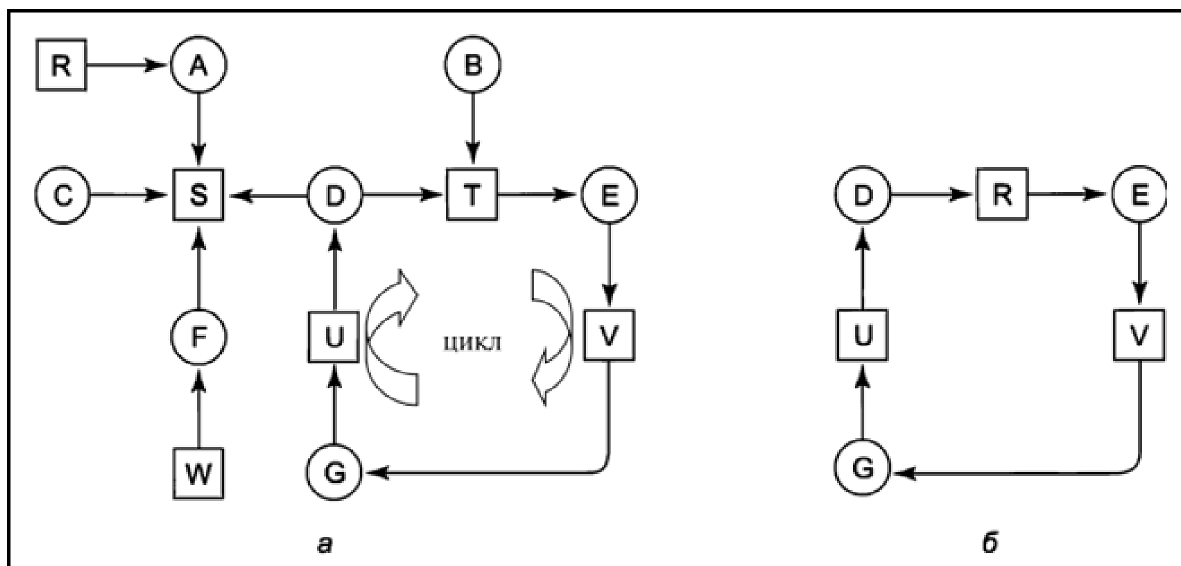


Рисунок 9.4 – Граф ресурсів і процесів *(а)*; витягнутий з нього цикл (*б*)

Хоча візуально виділити взаємне блокування з простого графа відносно неважко, для використання в справжніх системах потрібний формальний алгоритм виявлення взаємних блокувань. Відомо багато алгоритмів для виявлення циклів у спрямованих графах. Далі буде наведений простий алгоритм, який перевіряє граф і який припиняє свою роботу або при виявленні циклу, або при виявленні відсутності циклів. В алгоритмі використовується одна динамічна структура даних *L*, що є списком вузлів, а також списком ребер. У процесі роботи алгоритму ребра будуть відмічатися для позначення того, що вони вже були перевірені, щоб запобігти повторним перевіркам.

Дія алгоритму заснована на виконанні таких кроків:

1. Для кожного вузла N, наявного в графі, виконуються наступні п'ять кроків, що використовують вузол N в якості початкового.
2. Ініціалізувався (очищається) список L, а з усіх ребер знімаються позначки.

3. Поточний вузол додається до кінця списку L, і проводиться перевірка, чи не з'явиться цей вузол в списку L двічі. Якщо це станеться, значить, граф містить цикл (відображений в списку L) і алгоритм припиняє роботу.
4. Для заданого вузла визначається, чи немає яких-небудь непомічених ребер, що відходять від нього. Якщо такі ребра є, здійснюється перехід до кроку 5, якщо їх немає, здійснюється перехід до кроку 6.
5. Довільно вибирається і позначається непомічене ребро, що відходить від вузла. Потім по ньому здійснюється перехід до нового поточного вузла, і алгоритм повертається до кроку 3.
6. Якщо цей вузол є первинним вузлом, значить, граф не містить ніяких циклів, і алгоритм завершує свою роботу. Інакше алгоритм зайшов у тупик. Цей вузол видаляється, і алгоритм повертається до попереднього вузла, тобто до того вузла, який був поточним перед тільки що видаленим вузлом, цей вузол робиться поточним і здійснюється перехід до кроку 3.

Цей алгоритм бере по черзі кожен вузол в якості кореневого, в надії, що з цього вийде дерево, і виконує в дереві пошук у глибину. Якщо в процесі обходу алгоритм обходить усі ребра з якого-небудь заданого вузла і повертається до вузла, що вже зустрічався, значить, він знайшов цикл. Якщо він повертається до кореневого вузла і не може йти далі, то підграф, доступний з поточного вузла, не містить циклів. Якщо ця властивість зберігається для всіх вузлів, то це означає, що повний граф не містить циклів, а система не знаходиться в стані взаємного блокування.

Щоб побачити на практиці роботу цього алгоритму, скористаємося графом на рис. 9.4, *а*. Порядок обробки вузлів довільний, тому досліджуватимемо їх зліва направо і зверху вниз, вибравши при першому запуску алгоритму початковий вузол *R*, потім послідовно вибираючи вузли *A, B, C, S, D, T, E, F* і т. д. Якщо ми виявимо цикл, алгоритм припинить свою роботу.

Розпочинаємо з вузла *R* і ініціалізуємо *L* як порожній список. Потім додаємо вузол *R* в список, переходимо до єдино можливого вузла А і додаємо його також до списку *L,* отримуючи *L =[R, A]*. З вузла *A* йдемо до вузла *S,* отримуючи *L =[R, A, S]*. Вузол *S* не має ребер, що відходять від нього, отже, це тупик, який примушує нас повернутися до вузла *A*. Оскільки у вузла *A* також немає немаркованих ребер, що відходять від нього, ми повертаємося до вузла *R*, завершуючи, таким чином, його дослідження.

Тепер перезапускаємо алгоритм, розпочинаючи його роботу з вузла *A,* заздалегідь повернувши список *L* в початковий стан. Цей пошук також швидко зупиниться, тому розпочнемо знову з вузла *B*. З вузла *B* пройдемо по ребрах, що відходять, до тих пір, поки не дістанемося до вузла *D*. До цього моменту список матиме такий вигляд: *L =[B, T, E, V, G, U, D]*. Тепер треба зробити довільний вибір. Якщо вибрати вузол *S,* ми потрапляємо в тупик і повертаємося до вузла *D*. Удруге вибираємо вузол *T* і оновлюємо список *L* до виду *[B, **T,** E, V, G, U, D, **T**],* де виявляємо цикл і зупиняємо роботу алгоритму.

Цей алгоритм ще далекий від оптимального. Проте наведений приклад доводить саме існування алгоритму для виявлення взаємних блокувань.

9.5.2 Використання декількох ресурсів кожного типу

Коли в системі існує декілька примірників яких-небудь ресурсів, то для виявлення взаємних блокувань потрібний інший підхід. Зараз буде представлений алгоритм (алгоритм Медніка) [29], заснований на використанні матриць і призначений для виявлення взаємних блокувань при роботі *n* процесів, від P_1 *до* P_n. Нехай *m* – це число класів ресурсів, *E1* – кількість ресурсів класу *1, E2* – кількість ресурсів класу *2,* а загалом, *Ei* – кількість ресурсів класу *i* (де $1 \le i \le m$). *E* – це **вектор існуючих ресурсів.** Він передає загальну кількість примірників кожного ресурсу, що є в наявності. Наприклад, якщо клас 1 є накопичувачами на магнітних стрічках, то *E1 = 2* означає, що в системі є два такі накопичувачі.

У будь-який момент часу якісь ресурси можуть бути виділені і недоступні. Нехай *A* буде **вектором доступних ресурсів,** де *Ai* дає кількість примірників ресурсу *i,* доступних на даний момент (тобто не виділених). Якщо обидва накопичувачі на магнітній стрічці вже виділені, *A1* дорівнюватиме 0.

Тепер нам потрібні два масиви: *C* – **матриця поточного розподілу** і *R* – **матриця запитів,** *i*-й рядок в матриці *C* говорить про те, скільки примірників кожного класу ресурсів в даний момент утримує процес *Pi.* Таким чином, *Cij* – це кількість примірників ресурсу *j,* яка утримується процесом *i.* За аналогією з цим, *Rij* – це кількість примірників ресурсу *j,* яку хоче отримати процес *Pi*. Усі чотири структури даних показані на рис. 9.5.

Для цих чотирьох структур даних зберігається одне важливе співвідношення – кожен ресурс є або виділеним, або доступним. Це спостереження означає, що (*i = 1, 2,., n; j = 1,2, ., m*):

$$\sum_{i=1}^{n} Cij + Aj = Ej$$

Іншими словами, якщо скласти усі вже виділені екземпляри ресурсу *j* і до них додати все ще доступні примірники, в результаті вийде кількість існуючих примірників ресурсу цього класу.

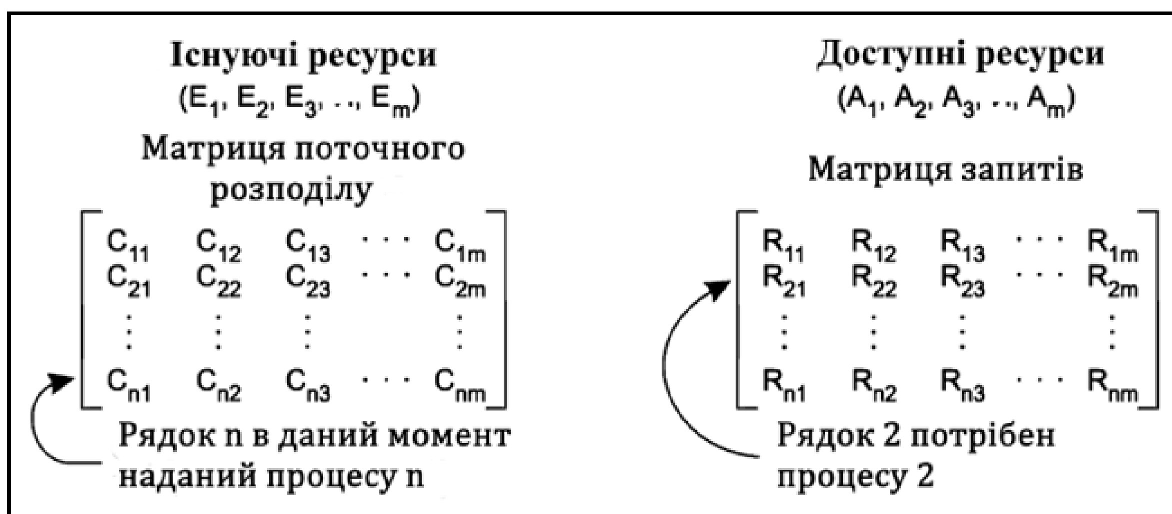


Рисунок 9.5 – Чотири структури даних, необхідні для роботи алгоритму виявлення взаємних блокувань

Алгоритм виявлення взаємних блокувань заснований на порівнянні векторів. Визначимо, що для двох векторів A і B відношення $A \leq B$ означає, що кожен елемент вектору A менше або дорівнює відповідному елементу вектору B. Математично це можна записати так: $A \leq B$ тоді і тільки тоді, коли $Ai \leq Bi$ для $1 \leq i \leq m$.

Кожен процес спочатку оголошується немаркованим. У міру роботи процеси позначатимуться, показуючи, що вони здатні завершити свою роботу і не беруть участь у взаємних блокуваннях. Коли алгоритм завершує свою роботу, будь-який немаркований процес вважається таким, що бере участь у взаємному блокуванні. При роботі цього алгоритму передбачається найгірший з можливих сценаріїв розвитку подій: усі процеси утримують усі отримані ресурси до тих пір, поки не закінчать свою роботу.

Тепер алгоритм виявлення взаємного блокування можна викласти в такій послідовності:

1. Пошук немаркованого процесу, *Pi,* для якого *i-й* рядок матриці *R* менше або рівний *A*.
2. Якщо такий процес знайдений, додаємо до вектору *A i-й* рядок матриці *C* (повертаємо зайняті ресурси), встановлюємо мітку на процес і повертаємося до кроку 1.
3. Якщо такого процесу немає, алгоритм завершує роботу.

Після закінчення роботи алгоритму усі немарковані процеси, якщо такі є, вважаються такими, що беруть участь у взаємних блокуваннях.

На першому кроці алгоритм шукає процес, який може допрацювати до кінця. Такий процес характеризується тим, що всі його запити на ресурси можуть бути задоволені за рахунок поточних доступних ресурсів. Тоді вибраний процес допрацює до кінця, після чого поверне всі утримувані ним ресурси до фонду доступних ресурсів. Потім цей процес позначається завершеним. Якщо в результаті виявиться, що всі процеси можуть допрацювати до кінця, що жоден з них не бере участі у взаємних блокуваннях. Якщо частина процесів ніколи не зможе допрацювати до кінця, значить, вони знаходяться в стані взаємних блокувань. Хоча алгоритм не є детермінованим (оскільки він може запускати процеси в будь-якому можливому порядку), результат завжди буде однаковий.

Розглянемо приклад роботи алгоритму виявлення взаємних блокувань, який показаний на рис. 9.6. Тут зображені три процеси і чотири класи ресурсів, які ми довільно позначили як диски, плотери, сканери і приводи компакт-дисків. Процес 1 утримує один сканер. Процес 2 утримує два диска і один привід компакт-дисків. Процес 3 утримує плотер і два сканери. Кожен процес потребує додаткових ресурсів, що відображено в матриці *R*.

Під час роботи алгоритму виявлення взаємних блокувань здійснюється пошук процесу, чий запит на ресурс може бути задоволений. Вимоги першого процесу задовольнити неможливо через відсутність доступного приводу компакт-дисків. Запит другого процесу також не можна задовольнити, оскільки немає вільного сканера. Можна задовольнити запит третього процесу, тому третій процес запускається і вивільняє всі ресурси, що утримувалися ним, внаслідок чого виходить, що ***A=(2 2 2 0).***

Рисунок 9.6 – Приклад, що демонструє роботу алгоритму виявлення взаємних блокувань

Тепер може бути запущений процес 2, що вивільняє утримувані ним ресурси, внаслідок чого виходить, що ***A=(4 2 2 1)*** і може бути запущений процес, що залишився. При цьому взаємних блокувань у системі не виникає.

Розглянемо незначну зміну ситуації, показаної на рис. 9.6. Припустимо, що процес 2 потребує приводу компакт-дисків, а також двох дисків і плотера. Жоден з цих запитів не може бути задоволений, тому уся система увійде до стану взаємного блокування.

Тепер, коли ми знаємо, як можна виявити взаємне блокування (принаймні, при заздалегідь відомих статичних запитах на виділення ресурсів), виникає питання, коли саме треба приступати до їх пошуку. Тобто, як часто слід виконувати перевірку тупика?

Можна перевірку при видачі кожного запиту на виділення ресурсу. Тим самим буде забезпечено їх виявлення на найранішій стадії, але це занадто обтяжливо для центрального процесора.

У деяких реалізаціях ОС застосовує ще «ледачішу» політику. Незадоволений запит може призвести до тупику, але може і не призвести. ОС не поспішає виконувати перевірку навіть при надходженні такого запиту, а вичікує деякий час: може бути «все само собою улагодиться» – і в більшості випадків саме так і трапляється. І тільки якщо запит залишається незадоволеним впродовж певного часу, ОС береться за пошук тупику. Розмір часової затримки може визначатися швидкісними характеристиками запрошеного ресурсу.

Є й інші альтернативні стратегії, що передбачають перевірку кожні k хвилин, або тільки в тому випадку, якщо міра завантаженості процесора знижується відносно якогось порогу.

Якщо тупик виявлений, то як його ліквідовувати? На жаль, розв'язка тупика практично завжди пов'язана з втратами. Єдиним реальним способом розв'язки є примусове припинення одного або декількох процесів і звільнення утримуваних ними ресурсів. Як вибрати процес-жертву для припинення його роботи?

По-перше, ОС має бути упевнена в тому, що при припиненні вибраних процесів звільниться об'єм ресурсів, достатній для виходу із тупика. По-друге, оцінюється об'єм втрат, пов'язаних з припиненням того або іншого процесу.

Припинений процес, швидше за все, буде запущений повторно. Таким чином, ресурси, використані ним при його незавершеному виконанні, складають прямі втрати. Тому природним рішенням видається припинення того процесу, який до цього моменту використав менше всього ресурсів (не лише монопольних, а будь-яких ресурсів взагалі). Оскільки найдорожчим ресурсом є процесорний час, то вибір жертви за критерієм мінімального використаного часу робиться найчастіше.

Бажано, щоб «постраждалий» процес був знову запущений, причому, можливо навіть, з підвищеним пріоритетом. Але чи всякий процес можна перервати на середині, а потім запустити спочатку?

Наприклад, нехай є процес-програма, яка повинна нарахувати внески на 10 банківських рахунків. Ця програма примусово завершується в той момент, коли вона обробила тільки 5 рахунків. Якщо при перезапуску програма почне виконуватися з початку, вона повторно нарахує внески на перші 5 рахунків. ОС не може знати, чи призведе перезапуск процесу до небажаних наслідків, тому рішення про повторний запуск перекладається на користувача.

9.5.3 Безпечний і небезпечний стан

При подальшому розгляді алгоритмів ухилення від взаємних блокувань використовується інформація, представлена на рис. 9.5. У будь-який заданий момент часу існує поточний стан структур *E, A, C* і *R*. Стан вважається ***безпечним***, якщо існує якийсь порядок планування, при якому кожен процес може допрацювати до кінця, навіть якщо усі процеси несподівано і терміново запросять максимальну кількість ресурсів. Це положення найпростіше проілюструвати за допомогою прикладу, в якому використовується один ресурс.

На рис. 9.7, *а* показаний стан, в якому процес *A* утримує 3 примірники ресурсу, але кінець кінцем може зажадати усі 9 примірників. Процес B у цей момент утримує 2 примірника, але пізніше може зажадати в цілому ще 4. Процес *C* також утримує 2 примірника, але може зажадати ще 5. У системі є всього 10 примірників цього ресурсу, 7 з яких вже розподілені, а 3 поки що вільні.

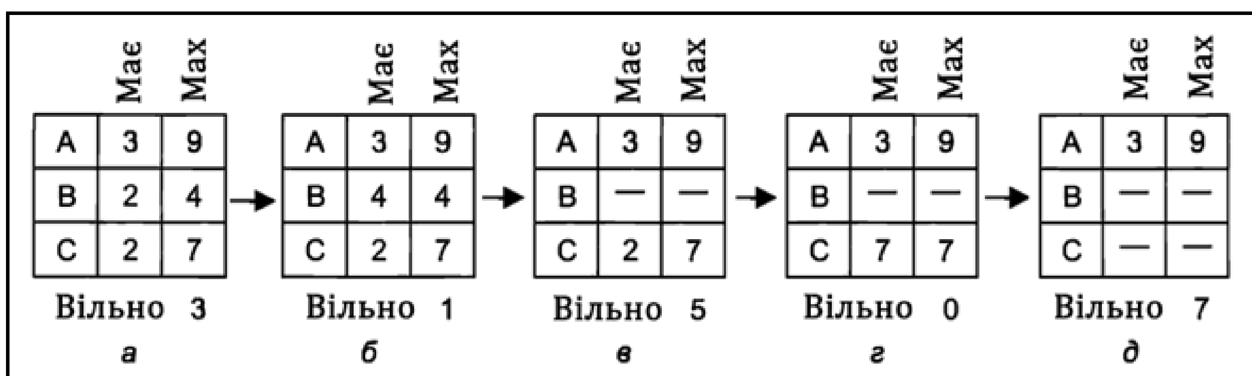


Рисунок 9.7 – Демонстрація того, що стан *(а)* є безпечним

Стан на рис. 9.7, *а* є безпечним, тому що існує така послідовність надання ресурсів, яка дозволяє завершитися усім процесам – планувальник може просто запустити в роботу тільки процес на той час, який він запросить і отримає два додаткові примірники ресурсу, що призведе до стану, зображеного на рис. 9.7, *б*. Коли процес *B* завершить свою роботу, ми отримаємо стан, показаний на рис. 9.7, *в*. Потім планувальник може запустити процес *С*, що з часом призведе нас до ситуації, показаної на рис. 9.7*, г*. Після закінчення роботи процесу *С* отримаємо ситуацію, показану на рис. 9.7*, д*.. Тепер процес *А* може отримати необхідні йому шість примірників ресурсу і завершити свою роботу. Таким чином, стан, показаний на рис. 9.7, *а* є безпечним, оскільки система може уникнути взаємних блокувань за допомогою ретельного планування процесів.

Тепер припустимо, що початковий стан системи показаний на рис. 9.8, *а* [25]. Але в даний момент процес *А* просить і отримує ще один ресурс, і система переходить в стан, показаний на рис. 9.8, *б*. Чи зможемо ми знайти послідовність, яка гарантує безпечну роботу системи?

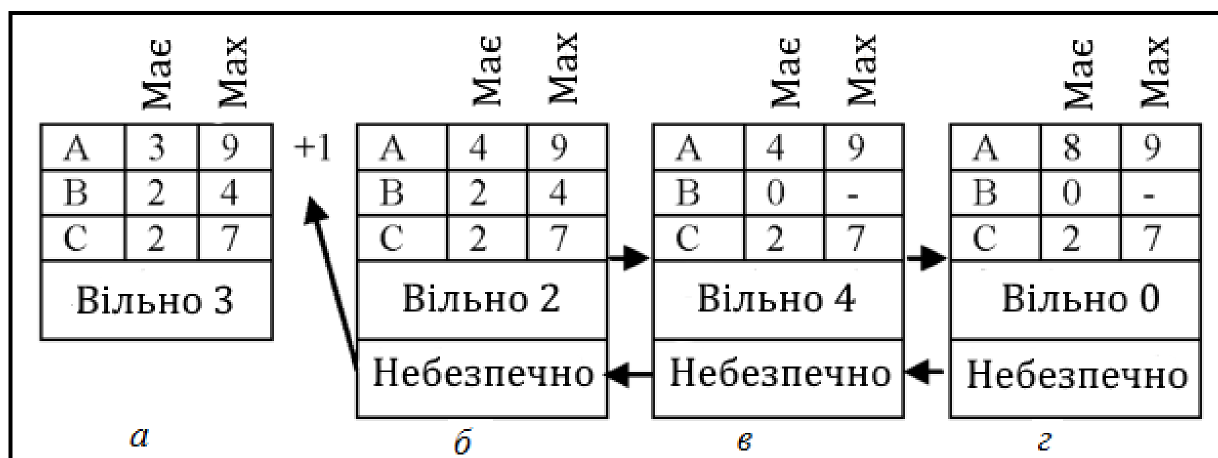


Рисунок 9.8 – Демонстрація того, що стан (*б*) небезпечний

Планувальник може дати попрацювати процесу *B* до того моменту, поки він не запросить усі свої ресурси, як показано на рис. 9.8, *c*. Зрештою процес *B* успішно завершується, і ми отримуємо ситуацію, показану на рис. 9.8*, г*. У результаті в системі залишилися тільки чотири вільні примірники ресурсу, а кожному з активних процесів необхідно по п'ять примірників.

Отже, рішення про надання ресурсу, яке перевело систему із стану, показаного на рис. 9.8, *а,* до стану, показаного на рис. 9.8*, б,* перевело її з безпечного в небезпечний стан. Якщо із стану, показаного на рис. 9.8*, б,* запустити процес *А* або процес *С,* то жоден з них не запрацює. Повертаючись назад, треба сказати, що запит процесу *А* не повинен був задовольнятися.

Слід зазначити, що небезпечний стан сам по собі не є станом взаємного блокування. Починаючи із стану, показаного на рис. 9.8, *б*, система може попрацювати деякий час. Фактично може навіть успішно завершитися робота одного з процесів. Таким чином, різниця між безпечним і небезпечним станами полягає в тому, що в безпечному стані система може гарантувати, що всі процеси закінчать свою роботу, а в небезпечному стані такої гарантії дати не можна.

9.5.4 Реалізація алгоритму виявлення тупика

У загальному випадку для побудови алгоритму виявлення тупика (алгоритм Медніка) використовуватимемо ті ж структури, що визначені вище [29]. Вектор *Available* (довжини *m*) містить інформацію про **доступні ресурси.** Якщо *Avaliable[j]=k,* то в системі в даний момент доступно *k* одиниць ресурсу *j.* Матриця *Allocation(n*m)* відображає **фактичне** виділення системою ресурсів процесам. Якщо *Allocation [i, j] = k,* то процесу *i* в даний момент виділено системою *k* одиниць ресурсу *j.*

Вектор *Requesti* (довжини *m*) – вектор запитів для процесу *Pi.* Якщо *Requesti [j] = k,* то процес *Pi* просить *k* екземплярів ресурсу *Rj*.

Алгоритм виявлення тупиків.

Крок 1. Ініціалізація.

Work = Available

Для *i = 1, ..., n*, якщо *Allocation [i] != 0,* то *finish [i] = false* інакше *finish [i] = true*.

Крок 2. Знаходимо *i*, таке, що:

Finish [i] = false

Request [i] <= Work

Якщо такого *i* немає, переходимо до кроку 4.

Крок 3.

Work = Work + Allocation [i]

Finish [i] = true

Перехід до кроку 2.

Крок 4. Якщо *Finish[i] = false* для деякого *i* від *1* до *n*, то система в стані тупика.

Більше того, якщо *Finish[i] = false,* то процес *Pi* – в стані тупика.

Обгрунтування і доказ коректності алгоритму надаємо читачеві.

Приклад використання алгоритму виявлення тупиків.

Нехай є 5 процесів – *P0*, …, *P4*, і 3 типи ресурсів – ресурс *A* (7 примірників), ресурс *B* (2 примірники) і ресурс *C* (6 примірників). Нехай стан системи в момент $\mathbf{T}_0$ такий:

	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	0
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2

У цьому стані системи послідовність процесів *<P0, P2, P3, P1, P4>* безпечна (перевірте це!).

У продовження прикладу, нехай процес *P2* просить додатковий ресурс типу ***C***:

	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

У даному випадку має місце тупик, в якому знаходяться процеси *P1, P2, P3, P4*. Перевірте це.

9.6 Відновлення після взаємного блокування

Припустимо, що наш алгоритм виявлення взаємного блокування успішно відпрацював і виявив таке блокування. Що ж далі? Потрібні якісь методи виходу з нього, що дозволяють системі відновити працездатність. У цьому розділі будуть розглянуті різні способи виходу із взаємного блокування.

Виявивши тупик, можна вивести з нього систему, порушивши одну з умов існування тупика. При цьому, можливо, декілька процесів частково або повністю втратять результати виконаної роботи. Складність відновлення обумовлена рядом чинників. У більшості систем немає досить ефективних засобів, щоб призупинити процес, вивести його із системи і відновити згодом із того місця, де він був зупинений. Додатково введемо робочий цілочисельний вектор *Work* (довжини *m)* і булевий вектор *Finish* (довжини *n)*. Вектор *Work* відображає пробні виділення ресурсів. Вектор *Finish* представляє інформацію про завершення процесів при цьому стані системи.

9.6.1 Відновлення за допомогою перерозподілу ресурсів

Іноді можна тимчасово відібрати ресурс у його поточного процеса і передати його іншому процесу. У більшості випадків для цього може знадобитися втручання оператора, особливо в операційних системах пакетної обробки, що запускаються на універсальних машинах.

Наприклад, щоб відібрати лазерний принтер у процеса, оператор може скласти усі вже віддруковані листи. Потім процес може бути призупинений (помічений як непрацездатний). Після цього принтер може бути виділений іншому процесу. Коли цей процес завершить свою роботу, стопка віддрукованих листів паперу може бути поміщена назад в приймальний лоток принтера і робота початкового процесу може бути відновлена.

Можливість відібрати ресурс у процесу, дозволити використати його іншому процесу, а потім повернути його без сповіщення процесу багато в чому залежить від природи цього ресурсу. Відновлення цим способом частенько ускладнене або зовсім неможливе. Вибір процесу для призупинення обумовлений тим, який саме процес має той ресурс, що у нього можна легко відібрати.

9.6.2 Відновлення шляхом відкату

У ряді систем реалізовані засоби відкату і перезапуску або рестарту процесу з контрольної точки (збереження стану системи в якийсь момент часу). Якщо проектувальники системи знають, що тупик імовірний, вони можуть періодично організовувати для процесів **контрольні точки**. Іноді це доводиться робити розробникам прикладних програм. Це означає, що стан процесу записується у файл, що дозволить здійснити його подальший перезапуск. Контрольні точки містять не лише образ пам'яті, але і стан ресурсів, тобто інформацію про те, які ресурси в даний момент виділені процесу. Для більшої ефективності нова контрольна точка повинна записуватися не поверх старої, а в новий файл, щоб під час виконання процесу зібралася ціла послідовність контрольних точок.

При виявленні взаємного блокування нескладно визначити, які ресурси потрібні. Щоб вийти із взаємного блокування, процес, що володіє необхідним ресурсом, відкачується назад до точки, передуючої отриманню цього ресурсу, для чого він запускається з однієї зі своїх контрольних точок. Уся робота, виконана після цієї контрольної точки, втрачається. Наприклад, має бути викинута вся віддрукована після цієї контрольної точки вихідна інформація, оскільки вона буде віддрукована знову. Фактично процес повертається до попереднього моменту, коли він ще не мав того ресурсу, який тепер виділений одному з тих, що беруть участь у взаємному блокуванні процесу. Якщо перезапущений процес намагається знову отримати ресурс, йому доводиться чекати, поки той не стане доступний.

9.6.3 Відновлення шляхом знищення процесів

Найгрубішим, але й найпростішим способом перервати взаємне блокування є знищення одного або декількох процесів. Можна знищити процес, що знаходиться в циклі взаємного блокування. Якщо повезе, то інші процеси зможуть продовжити свою роботу. Якщо це не допоможе, то все можна повторити, поки цикл не буде розірваний.

В якості альтернативи жертвою можна обрати процес, що не знаходиться в циклі, щоб він вивільнив утримувані ним ресурси. При цьому підході знищуваний процес обирається з особливою ретельністю, оскільки він повинен утримувати ресурси, необхідні деяким процесам в циклі. Наприклад, один процес може утримувати принтер і вимагати плотер, а інший процес, навпаки, утримує плотер і просить принтер. Обидва вони знаходяться в стані взаємного блокування. Третій процес може утримувати інший такий же принтер і інший такий же плотер і успішно працювати. Знищення третього процесу призведе до вивільнення цих ресурсів і зруйнує взаємне блокування перших двох процесів.

По можливості краще убити процес, який може бути безболісно перезапущений із самого початку. Наприклад, компіляція завжди може бути перезапущена, оскільки все, що вона робить – це читає вхідний файл і створює об'єктний файл. Якщо процес компіляції буде знищений на півдорозі, то перший запуск не вплине на другий.

А ось процес, що оновлює базу даних, не завжди можна буде безпечно запустити вдруге. Якщо процес додає одиницю до якого-небудь запису таблиці бази даних, то його первинний запуск, знищення, а потім повторний запуск призведуть до невірного результату, оскільки до поля буде додана двійка.

9.7 Ухилення від взаємних блокувань

Мета запобігання взаємних блокувань – забезпечити умови, що унеможливлюють виникнення тупикових ситуацій. Система повинна вміти приймати рішення про те, чи представляє виділення ресурсу небезпеку, чи ні, і виділяти його тільки в тому випадку, коли це безпечно. Розглянемо способи попередження взаємних блокувань за рахунок ретельного розподілу ресурсів.

9.7.1 Заборона запуску процесу

Розглянемо систему з *n* процесів і *m* різних типів ресурсів. З рис. 9.5 ми маємо такі вектори і матриці:

Ресурс =	$(E1, E2...Em)$	Загальна кількість кожного ресурсу в системі
Доступність =	$(A1, A2...Am)$	Загальна кількість кожного ресурсу, не виділеного процесам
Вимоги =	$\begin{pmatrix} R11, R12...R1m \\ R21, R22...R2m \\ Rn1, Rn2...Rnm \end{pmatrix}$	Запити кожного процесу на кожен з ресурсів
Розподіл =	$\begin{pmatrix} C11, C12...C1m \\ C21, C22...C2m \\ Cn1, Cn2...Cnm \end{pmatrix}$	Поточний розподіл ресурсів

Матриця вимог (запитів), в якій кожен рядок описує один з процесів, вказує максимальні вимоги кожного процесу до різних ресурсів, тобто *Rij* – це вимоги процесом *i* ресурсу *j.* Для забезпечення працездатності методу усунення взаємоблокувань ця інформація має бути оголошена процесом заздалегідь. Аналогічно*, Cij* – поточна кількість розподіленого ресурсу *j*, виділене процесу *i.*

Повинні виконуватися такі співвідношення:

1. $Ei = Ai + \sum_{k=1}^{n} Cki$ для всіх *i:* усі ресурси або вільні, або виділені.
2. $Rki \le Ei$, для всіх *k* і *i:* жоден процес не може затребувати ресурс, що перевищує його загальну кількість в системі.
3. $Cki \le Rki$ для всіх *k* і *i*: жоден процес не може отримати більше ресурсів, ніж ним було затребувано.

Коли усі вказані величини визначені, ми в змозі створити стратегію усунення взаємних блокувань, яка забороняє запуск нового процесу, якщо його вимоги ресурсів можуть призвести до взаємного блокування. Новий процес P_{n+1} запускається тільки якщо

$$Ei \geq R(n+1)i + \sum_{k=1}^{n} Rki \text{ для всіх } i.$$

Це означає, що запуск нового процесу станеться тільки тоді, коли можуть бути задоволені максимальні вимоги усіх поточних процесів плюс вимоги процесу, що запускається. Ця стратегія аж ніяк не є оптимальною, оскільки припускає гірше: що усі процеси пред'являть максимальні вимоги одночасно.

9.7.2 Алгоритм банкіра для одного ресурсу

Можна уникнути взаємного блокування, якщо розподіляти ресурси, дотримуючись певних правил. Серед такого роду алгоритмів найвідоміший алгоритм банкіра, який запропонував Дейкстра (Dijkstra, 1965 р., уперше реалізований в операційній системі THE в кінці 1960-х рр.), який базується на ***безпечних*** або ***надійних*** станах (safe state) [14]**.** Безпечний стан – це такий стан, для якого є, принаймні, одна послідовність подій, яка не призведе до взаємного блокування. Модель алгоритму заснована на діях банкіра, який, маючи в наявності капітал, видає кредити.

Суть алгоритму полягає в наступному. Алгоритм перевіряє, чи веде виконання кожного запиту до небезпечного стану. Якщо так, то запит відхиляється. Якщо задоволення запиту до ресурсу призводить до безпечного (надійному) стану, ресурс надається процесу. На рис. 9.9, *а* показані чотири клієнти: *A, B, C* і *D,* кожен з яких отримав певну кількість одиниць кредиту. Банкір знає, що не всім клієнтам миттєво знадобиться максимальна сума їх кредиту, тому для обслуговування їх потреб він зарезервував тільки 10 одиниць, а не всі 22, які потрібні клієнтам. Щоб провести аналогію з комп'ютерною системою, вважатимемо, що клієнти – це процеси, одиниці – накопичувачі на дисках, а банкір – це операційна система.

Клієнти займаються своїми справами, просячи час від часу позики (тобто, ресурси, по одному за один запит). Процесам дозволяється утримувати за собою ресурси, просячи і чекаючи виділення додаткових ресурсів. Система може або задовольнити, або відхилити кожен запит. В якийсь певний момент виникає ситуація, показана на рис. 9.9, *б.* Цей стан не представляє небезпеки, оскільки при двох одиницях, що залишилися, банкір може відкласти виконання будь-яких запитів, за винятком запиту клієнта *C,* дозволяючи *C* завершити свої справи і вивільнити усі чотири ресурси. Маючи у своєму розпорядженні чотири ресурси, банкір може дозволити отримати необхідні одиниці або *D* або *B* тощо.

Розглянемо, що вийде, якщо запит від *B* однієї додаткової одиниці буде задоволений в ситуації, показаній на рис. 9.9, *б.* Ми отримаємо небезпечну ситуацію, показану на рис. 9.9, *в.* Якщо усі клієнти несподівано запросять свої максимальні позики, банкір не зможе задовольнити нікого з них, і ми отримаємо взаємоблокування. Небезпечний стан не обов'язково призводить до взаємоблокування, оскільки клієнтові може не знадобитися уся доступна максимальна сума кредиту, але банкір не може розраховувати на це.

	Має	Max
A	0	6
B	0	5
C	0	4
D	0	7

Вільно : 10

а

	Має	Max
A	1	6
B	1	5
C	2	4
D	4	7

Вільно : 2

б

	Має	Max
A	1	6
B	2	5
C	2	4
D	4	7

Вільно : 1

в

Рисунок 9.9 – Стани розподілу ресурсів: безпечне *(а)*, безпечне (б), небезпечне *(в)*

9.7.3 Алгоритм банкіра для декількох типів ресурсів

Алгоритм банкіра може бути поширений на роботу з декількома ресурсами. На рис. 9.10 показано, як він працює.

Процес	Диски	Плотери	Сканери	Компакт-диски
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

а) Розподілені ресурси

Процес	Диски	Плотери	Сканери	Компакт-диски
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

E = (6342)
P = (5322)
A = (1020)

б) Ресурси, які ще потрібні

Рисунок 9.10 – Алгоритм банкіра для системи з декількома типами ресурсів

На рисунку зображені дві матриці. Ліва матриця (*а*) показує, скільки екземплярів кожного ресурсу в даний момент виділені кожному з п'яти процесів. Права матриця (*б*) показує, скільки екземплярів ресурсів все ще потрібні кожному процесу для завершення його роботи. На рис. 9.6 ці матриці називалися C і R. Як і у випадку з ресурсом одного типу, процеси перед виконанням своєї роботи повинні повідомити про свої загальні потреби в ресурсах, щоб система в будь-який момент могла обчислити праву матрицю (*б*).

Три вектори, що зображені праворуч від матриць, показують відповідно існуючі ресурси (вектор E), зайняті ресурси (вектор P) і доступні ресурси (вектор A). Судячи зі значення вектору E, в системі є шість дисків, три плотери, чотири

сканери і два приводи компакт-дисків. З них зайняті в даний момент п'ять накопичувачів на дисках, три плотери, два сканери і два приводи компакт-дисків. Цей факт можна встановити шляхом складання значень чотирьох стовпців, що відповідають ресурсам, в лівій матриці. Вектор доступних ресурсів – це різниця між кількістю присутніх в системі ресурсів, і кількістю ресурсів використовуваних нині.

Суть алгоритму:

1. Припустимо, що в системі в наявності *n* пристроїв, наприклад сканерів.
2. ОС приймає запит від призначеного для користувача процесу, якщо його максимальна потреба не перевищує *n*.
3. Користувач гарантує, що якщо ОС в змозі задовольнити його запит, то усі пристрої будуть повернені системі впродовж кінцевого часу.
4. Поточний стан системи називається ***надійним***, якщо ОС може забезпечити усім процесам їх виконання впродовж кінцевого часу.
5. Відповідно до алгоритму банкіра виділення пристроїв можливе, тільки якщо стан системи залишається надійним.

Тепер може бути викладений алгоритм перевірки стану на безпеку:

1. Шукаємо в матриці *R* рядок, що відповідає процесу, чиї незадоволені потреби в ресурсах менше або дорівнюють вектору *A*. Якщо такого рядка не існує, то система врешті-решт увійде до стану взаємного блокування, оскільки жоден процес не зможе допрацювати до успішного завершення (передбачається, що процеси утримують усі ресурси, поки не завершать свою роботу).
2. Допускаємо, що процес, чий рядок був вибраний, просить усі необхідні йому ресурси (можливість чого гарантується) і завершує свою роботу. Відмічаємо цей процес як завершений і додаємо усі його ресурси до вектора *A*.
3. Повторюємо кроки 1 і 2 до тих пір, поки або усі процеси будуть помічені як завершені (у цьому випадку початковий стан може вважатися безпечним), або не залишиться процесів, чиї запити можуть бути задоволені (в цьому випадку виникне взаємне блокування).

Якщо в кроці 1 підходять для вибору декілька процесів, то неважливо, який з них буде вибраний: фонд доступних ресурсів або збільшується, або в гіршому разі залишається таким же.

Тепер повернемося до прикладу, показаному на рис. 9.10. Поточний стан безпечний. Припустимо, що процес *B* тепер зробив запит на сканер. Цей запит може бути задоволений, оскільки стан, що виходить в результаті, як і раніше безпечний (процес *D* може завершити свою роботу, потім це ж може зробити процес *A* або процес *E*, а потім і усі інші).

Уявімо тепер, що після виділення процесу *B* одного з двох сканерів, що залишилися, процес *E* зажадає останній сканер. Задоволення цього запиту зменшить значення вектору доступних ресурсів до *A(1 0 0 0),* що призведе до взаємоблокування. Ясно, що запит процесу *E* має бути на деякий час відхилений.

9.7.4 Реалізація алгоритму банкіра

Структури даних для алгоритму банкіра. Нехай в системі є *n* процесів і *m* типів ресурсів. Вектор *Available* довжини *m* містить інформацію про **доступні** ресурси. Якщо *Avaliable[j] = k,* то в системі в даний момент доступно *k* одиниць ресурсу *j*.

Матриця *Max(n*m)* відображає **максимальні** потреби процесів в ресурсах. Якщо *Max [i, j] = k,* то процес *i* може запросити не більше *k* одиниць ресурсу *j*.

Матриця *Allocation(n*m)* відображає **фактичне** виділення системою ресурсів. Якщо *Allocation [i, j] = k*, то процесу *i* в даний момент виділено системою *k* одиниць ресурсу *j*.

Матриця *Need(n*m)* відображає потреби процесів в ресурсах., які ще **залишилися,** Якщо *Need [i, j] = k,* то процесу *i* можуть знадобитися ще *k* одиниць ресурсу *j* для завершення роботи.

Має місце таке співвідношення між елементами матриць:

Need [i, j] = Max [i, j] – Allocation [i, j].

Алгоритм перевірки стану системи на безпеку. У позначеннях «Структури даних для алгоритму банкіра», розглянемо алгоритм перевірки стану системи на те, чи є він безпечним.

Введемо цілочисельний вектор *Work* (довжини *m*) і булевий вектор *Finish* (довжини *n*). Вектор *Work* відображає пробні виділення ресурсів. Вектор *Finish* представляє інформацію про завершення процесів при цьому стані системи.

Алгоритм безпеки.

Крок 1. Ініціалізація.

Work = Available

Finish [i] = false для i = 1, …, n.

Тут і надалі усі привласнення і порівняння, в яких беруть участь вектори або матриці, виконуються **поелементно**.

Крок 2. Знаходимо *i*, таке, що:

Finish [i] = false

Need [i] <= Work

Якщо такого *i* немає, переходимо до кроку 4.

Крок 3.

Work = Work + Allocation [i]

Finish [i] = true

Перехід до кроку 2.

Крок 4. Якщо *Finish[i] = true* для усіх *i = 1, ..., n,* то система в **безпечному стані**.

Необхідні пояснення до алгоритму:

1. Алгоритм будує безпечну послідовність номерів процесів *i,* якщо вона існує. На кожному кроці, після виявлення чергового елементу послідовності, алгоритм моделює звільнення *i-м* процесом ресурсів після його завершення.
2. На кроці 1 привласнення векторів виконується поелементно.

3. На кроці 2, *Need* – матриця потреб *(n * m); Need[i]* – рядок матриці, що представляє вектор потреб (довжини *m*) процесу *i*. Порівняння виконується поелементно, і його результат вважається істинним, якщо співвідношення виконане для всіх елементів векторів. Умова, що перевіряється, означає, що потреби процесу *i* зі знайденим номером можуть бути задоволені негайно, і процес може отримати їх і завершитися.
4. На кроці 3, *Allocation [i]* – рядок матриці *Allocation*, що означає поточні ресурси, виділені процесу *i.* За допомогою вектора *Work* моделюється звільнення ресурсів *i-м* процесом, після чого процесу привласнюється ознака завершеності.

Алгоритм запиту ресурсів для процесу *Pi* – основна частина алгоритму банкіра. Для основного алгоритму введемо вектор *Requesti* (довжини *m*) – вектор запитів для процесу *Pi*. Якщо *Requesti [j] = k*, то процес *Pi* просить *k* екземплярів ресурсу *Rj*.

Крок 1. Якщо *Requesti <= Need[i]*, перейти до кроку 2.
Інакше – згенерувати виняткову ситуацію
(процес перевищив свої максимальні потреби).

Крок 2. Якщо *Requesti <= Available*, перейти до кроку 3.
Інакше процес повинен чекати, оскільки ресурс недоступний.

Крок 3. Спланувати виділення ресурсу процесу *Pi*, модифікуючи стан системи таким чином:

Available = Available - Requesti
Allocation = Allocation + Requesti
Need [i] = Need [i] - Requesti

Викликати алгоритм перевірки безпеки отриманого стану.
Якщо стан безпечний, виділити ресурс процесу *Pi*. Вихід.
Якщо стан небезпечний, відновити попередній стан; процес повинен чекати.

Приклад використання алгоритму банкіра. Нехай є 5 процесів **–** *P0*, …, P4, і 3 типи ресурсів – ресурс А (10 екземплярів), ресурс В (5 екземплярів) і ресурс С (7 екземплярів). Нехай стан системи в момент T_0 такий:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Обчислимо матрицю потреб *Need = Max - Allocation*:

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Неважко бачити, що система знаходиться в безпечному стані. Послідовність процесів <*P1, P3, P4, P2, P0*> задовольняє критерію безпеки.

У продовження прикладу припустимо, що процес *P1* зробив запит (1 0 2). Перевіряємо, що *Request <= Available: <(1 0 2) <= (3 3 2) = true*.

Задовольняємо запит. Стан системи набирає вигляду:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	1	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Виконання алгоритму безпеки показує, що послідовність процесів <*P1, P3, P4, P0, P2*> задовольняє критерію безпеки.

9.7.5 Недоліки алгоритму банкіра

Хоча алгоритм чудовий у теорії, на практиці він, по суті, марний, оскільки нечасто можна визначити заздалегідь, які будуть максимальні потреби процесів в ресурсах. Крім того, кількість процесів не фіксована, вона динамічно змінюється в міру входу користувачів в систему і виходу їх з неї. І, більше того, ресурси, що вважалися доступними, можуть несподівано пропасти (плотер може зламатися). Таким чином, на практиці лише небагато систем, якщо такі взагалі є, використовують алгоритм банкіра для ухилення від взаємних блокувань.

Алгоритм банкіра представляє для нас інтерес тому, що він дає можливість розподіляти ресурси так, щоб обходити тупикові ситуації. Проте у цього алгоритму є ряд серйозних недоліків, із-за яких розробник системи може виявитися змушеним вибрати інший підхід до вирішення проблеми тупиків.

1. Алгоритм банкіра виходить з фіксованої кількості розподілюваних ресурсів. Оскільки пристрої, що представляють ресурси, частенько вимагають технічного обслуговування або із-за виникнення несправностей, або з метою профілактики, ми не можемо вважати, що кількість ресурсів завжди залишається фіксованою.

2. Алгоритм вимагає, щоб число працюючих користувачів (процесів) залишалося постійним. Подібна вимога також є нереалістичною. У сучасних мультипрограмних системах кількість працюючих користувачів увесь час міняється. Наприклад, велика система з розподілом часу цілком може обслуговувати 100 або більше користувачів одночасно. Проте поточне число обслуговуваних користувачів безперервно міняється, можливо, дуже часто, кожні декілька секунд.
3. Алгоритм вимагає, щоб клієнти (тобто завдання або процеси) гарантовано «платили борги» (повертали виділені їм ресурси) впродовж деякого кінцевого часу. І знову-таки для реальних систем потрібно набагато конкретніші гарантії.
4. Алгоритм вимагає, щоб користувачі заздалегідь вказували свої максимальні потреби в ресурсах. У міру того як розподіл ресурсів стає усе більш динамічним, все важче оцінювати максимальні потреби користувача.

9.8 Запобігання взаємних блокувань

Як все-таки можна уникнути взаємних блокувань в реальних системах? Відхилитися від них по суті неможливо, оскільки для цього потрібна інформація про майбутні запити, про які нічого не відомо. Щоб відповісти на це питання, повернемося назад до чотирьох умов, сформульованих Коффманом та іншими, і подивимося, чи зможуть вони дати нам ключ до вирішення проблеми. Якщо ми зможемо гарантувати, що хоч би одну з цих умов ніколи не буде виконано, то взаємні блокування стануть структурно неможливими.

9.8.1 Порушення умови взаємних блокувань

Доступ до деяких ресурсів має бути винятковим. Проте, деякі пристрої вдається усуспільнити. Як приклад розглянемо принтер. Відомо, що намагатися здійснювати виведення на принтер можуть декілька процесів. Щоб уникнути хаосу організовують проміжне формування усіх вихідних даних процесу на диску, тобто пристрої, що розділяється. Лише один системний процес, що називається сервісом або демоном принтера, який відповідає за виведення документів на друк у міру звільнення принтера, реально з ним взаємодіє. Ця схема називається **спулінгом** (spooling). Таким чином, принтер стає пристроєм, що розділяється, і тупик для нього усунений.

Що вийде, якщо кожен з двох процесів заповнить по половині доступного простору, виділеного на диску під черги на друк своїми вихідними даними, і жоден з них не сформує свої повні вихідні дані? У такому разі ми отримаємо два процеси, що завершили формування тільки частини, але не усього об'єму своїх вихідних даних, і не мають можливості продовжити свою роботу. Жоден з процесів не зможе коли-небудь завершитися, і ми отримаємо взаємне блокування, пов'язане з виведенням даних на диск. Але за наявності дисків великої місткості вичерпання дискового простору стає малоймовірним.

9.8.2 Порушення умови утримання і очікування ресурсів

Умови очікування ресурсів можна уникнути, зажадавши виконання стратегії двофазного захоплення. У першій фазі процес повинен просити усі необхідні йому ресурси відразу. До тих пір, поки вони не надані, процес не може продовжувати виконання.

Якщо в першій фазі деякі ресурси, які були потрібні цьому процесу, вже зайняті іншими процесами, то цей процесс повинен звільнити усі ресурси, які були йому виділені, і намагатися повторити першу фазу.

Цей підхід нагадує вимогу захоплення усіх ресурсів заздалегідь. Природно, що тільки спеціально організовані програми можуть бути призупинені впродовж першої фази і рестартовани згодом.

Таким чином, один із способів – змусити усі процеси зажадати потрібні їм ресурси перед виконанням («все або нічого»). Якщо система в змозі виділити процесу все необхідне, він може працювати до завершення. Якщо хоч би один з ресурсів зайнятий, процес чекатиме.

Це рішення застосовується в пакетних системах, які вимагають від користувачів перелічити всі необхідні його програмі ресурси. Проте в цілому подібний підхід не занадто привабливий і призводить до неефективного використання комп'ютера. Як уже відзначалося, перелік майбутніх запитів до ресурсів нечасто вдається спрогнозувати. Якщо така інформація є, то можна скористатися алгоритмом банкіра. Відмітимо також, що описуваний підхід суперечить парадигмі модульності в програмуванні, оскільки додаток повинен знати про передбачувані запити до ресурсів у всіх модулях.

9.8.3 Порушення принципу відсутності перерозподілу

Якби можна було відбирати ресурси в процесів до завершення їх роботи, то вдалося б добитися невиконання третьої умови виникнення взаємних блокувань. Перелічимо мінуси цього підходу.

По-перше, відбирати в процесів можна тільки ті ресурси, стан яких легко зберегти, а пізніше відновити, наприклад стан процесора. По-друге, якщо процес впродовж деякого часу використовує певні ресурси, а потім звільняє ці ресурси, то він може втратити результати роботи, виконаної за цей час. Нарешті, наслідком цієї схеми може бути дискримінація окремих процесів, в яких постійно відбирають ресурси.

Питання в ціні подібного рішення, яка може бути занадто високою, якщо необхідність відбирати ресурси виникає часто.

9.8.4 Порушення умови кругового очікування

Важко запропонувати розумну стратегію, щоб уникнути останньої умови з розділу «Умови виникнення тупиків» – циклічного очікування.

Один із способів уникнути умови кругового очікування – діяти відповідно до правила, згідно з яким кожен процес може мати тільки один ресурс в кожен

момент часу. Якщо потрібний другий ресурс – звільни перший. Очевидно, що для багатьох процесів це неприйнятно.

Інший спосіб – упорядкувати ресурси. Наприклад, можна присвоїти усім ресурсам унікальні номери і зажадати, щоб процеси просили ресурси в порядку їх зростання. Тоді кругове очікування виникнути не може. Наприклад, якщо процес запросив ресурс *R1*, то далі він може запросити тільки такі ресурси, які упорядкувани за значенням *R1*. Якби в прикладі, наведеному на рис. 9.1, потоки *A* і *B* замовляли ресурси в однаковому порядку (порт, диск), то взаємного блокування можна було б уникнути.

Нехай, наприклад, всі ресурси повністю впорядковані від **1** до ***r***. Ми можемо накласти наступне обмеження: процес не може запитувати ресурс ***Rk***, якщо він утримує ресурс ***Rh*** і при цьому ***k*** <***h***.

Видно, що, використовуючи це правило, ми ніколи не будемо входити в тупики. Наведемо приклад того, як застосовується це правило.

Нехай є процес, який використовує ресурси, впорядковані як *A, B, C, D, E*, в такий спосіб (рис. 9.11):

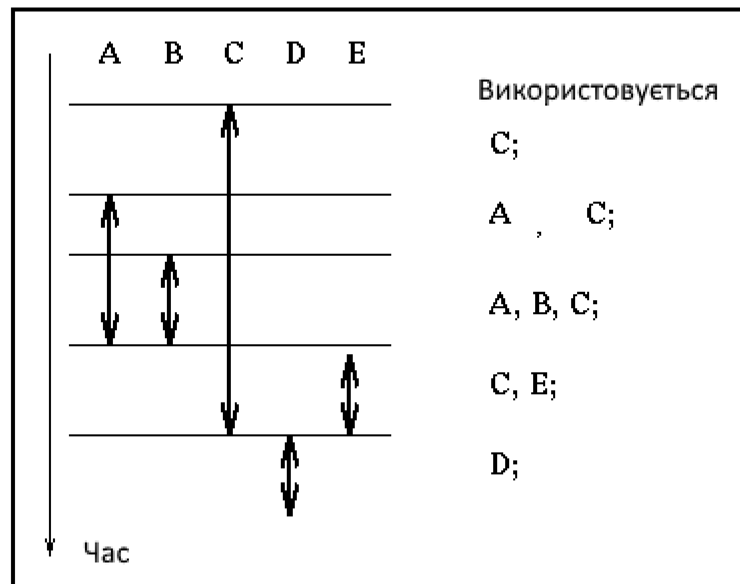


Рисунок 9.11 – Приклад впорядкування ресурсів

Тоді процес може робити наступне:

- *захопити (A); захопити (B); захопити (C);*
- *використати C;*
- *використати A, C;*
- *використати A, B, C;*
- *звільнити (A), звільнити (B), захопити (E);*
- *використати C і E;*
- *звільнити (C), звільнити (E), захопити (D);*
- *використати D;*
- *звільнити (D).*

Стратегія цього типу може використовуватися, коли ми маємо декілька ресурсів.

9.9 Зависання

Проблемою, тісно пов'язаною із взаємним блокуванням, є ***зависання***. У динамічній системі запит ресурсів відбувається постійно. Для того щоб прийняти рішення, хто, коли і який ресурс отримає, потрібна певна політика. Ця політика, навіть і розумна, може призвести до того, що деякі процеси ніколи не будуть обслужені, навіть якщо вони не знаходяться в стані взаємного блокування.

Як приклад розглянемо розподіл принтера. Уявимо собі, що система використовує деякий алгоритм, який гарантує, що розподіл принтера не призводить до взаємоблокування. Тепер припустимо, що декілька процесів разом захотіли отримати принтер у своє розпорядження. І хто його отримає?

Один з можливих алгоритмів передбачає передачу принтера тому процесу, в якого, наприклад, найменший файл для виведення на друк. Такий підхід до максимуму збільшує число щасливих клієнтів і представляється цілком справедливим. А тепер подивимося, що вийде на працюючій системі, де в одного процесу є для виведення на друк величезний файл. Як тільки принтер звільниться в черговий раз, система може вибрати знову процес з найкоротшим файлом. Якщо потік процесів з короткими файлами не вичерпується, процес з величезним файлом не отримає принтер ніколи. Він просто намертво зависне (буде відкладений назавжди, навіть якщо і не буде заблокований).

Зависання можна уникнути за рахунок використання політики розподілу ресурсів «першим прийшов – першим і обслужений». При такому підході процес, який очікує довше всіх, обслуговується наступним. Зрештою, будь-який заданий процес з часом стане найстаршим у черзі і отримає необхідний йому ресурс.

9.10 Тупики в системах спулінгу

Системи спулінга часто виявляються схильні до тупиків. Режим спулінга (введення-виведення з буферизацією) застосовується для підвищення продуктивності системи шляхом ізолювання програми від такого низькошвидкісного периферійного устаткування, як пристрій виведення даних на принтер. Якщо, наприклад, програмі, що видає рядки даних на принтер, доводиться чекати роздруку кожного рядка перед передачею наступного рядка, то така програма виконуватиметься повільно. Щоб підвищити швидкість виконання програми, рядки даних, призначені для друку, спочатку записуються на більше високошвидкісний пристрій, наприклад дисковий накопичувач, де вони тимчасово зберігаються до моменту роздруку.

У деяких системах спулінга програма повинна сформувати усі вихідні дані – тільки після цього починається реальний роздрук. У зв'язку з цим декілька незавершених завдань, що формують рядки даних і записують їх у файл спулінга для друку, можуть опинитися в тупиковій ситуації, якщо передбачена місткість буфера буде заповнена до того, як завершиться виконання якого-небудь завдання. Для відновлення, або виходу з подібної тупикової ситуації, міг би знадобитися перезапуск, рестарт системи з втратою всієї роботи, виконаної до цього моменту.

Якщо система потрапляє в тупикову ситуацію таким чином, що в оператора ЕОМ залишаються можливості управління, то в якості менш радикального способу відновлення працездатності можна запропонувати знищення одного або декількох завдань, поки у завдань, що залишаються, не виявиться достатньо вільного місця в буфері, щоб вони могли завершитися.

Коли системний програміст робить генерацію (чи налаштування) операційної системи, він задає розмір буферних файлів для спулінга. Один із способів зменшити ймовірність тупика при спулінгу полягає в тому, щоб передбачити значно більше місця для файлів спулінга, чим знадобиться згідно з попередньою оцінкою. Подібне рішення не завжди здійсненно, якщо пам'ять дефіцитна.

Поширеніше рішення полягає в тому, що для процесів вхідного спулінга встановлюються обмеження, з тим щоб вони не могли приймати додаткові завдання, коли файли спулінга починають наближатися до деякого порогу насичення, наприклад, виявляються заповненими на 75 відсотків. Такий підхід може призвести до деякого зниження продуктивності системи, проте це – та ціна, яку доводиться платити за зменшення ймовірності тупика.

Сучасні системи є в цьому сенсі набагато досконалішими. Вони можуть дозволяти починати роздрук до того, як завершиться чергове завдання, з тим щоб повний або майже повний файл спулінга спустошився повністю або частково вже в процесі виконання завдання. У багатьох системах передбачається динамічний розподіл буферної пам'яті, так що, якщо відведеного місця в пам'яті виявляється мало, для файлів спулінга виділяється додаткова пам'ять.

Контрольні питання і тести до розділу 9

Контрольні питання

1. Назвіть синоніми терміну «взаємне блокування».
2. Дайте визначення вивантажуваним і невивантажуваним ресурсам.
3. Які задачі включає проблема тупиків?
4. Які чотири необхідні умови повинні виконуватися для того, щоб взаємне блокування стало можливим?
5. Які структури даних використовуються алгоритмами для виявлення взаємного блокування?
6. Дайте визначення безпечному і небезпечному стану процесів.
7. Як відбувається відновлення системи після взаємного блокування за допомогою перерозподілу ресурсів?
8. Як відбувається відновлення системи після взаємного блокування шляхом відкату процесу до контрольної точки?
9. Які структури даних використовуються алгоритмом для заборони запуску процесу?
10. На яких умовах (станах) базується алгоритм банкіра, який запропонував Дейкстра?
11. Перелічить недоліки алгоритму банкіра.

12. Чи можна за допомогою впорядкування ресурсів порушити умову кругового очікування ресурсів?
13. Що загального між взаємним блокуванням і нескінченним відкладанням?
14. Яке обмеження накладається на процеси алгоритмом банкіра?
15. Чи правда що система не може перейти з ненадійного стану назад в надійний?
16. Чому виявити тупикову ситуацію в розподіленій системі набагато складніше, ніж на окремому комп'ютері?

Тести

1. Алгоритм Банкіра призначений для:
   1) виходу з тупика;
   2) виявлення тупика;
   3) відновлення після тупика;
   4) запобігання тупику.
2. Алгоритм Медніка забезпечує:
   1) запобігання тупикам;
   2) вихід з тупика;
   3) виявлення тупика;
   4) відновлення після тупика.
3. Виконання яких умов є необхідним і достатнім для утворення тупика?
   1) виконання усіх перерахованих умов;
   2) умова утримання і очікування;
   3) умова невивантажуваності (відсутності перерозподілу);
   4) умова кругового (циклічного) очікування;
   5) умова взаємовиключення (mutual exclusion).
4. Який підхід вирішення проблеми тупика прийнятий в більшості популярних ОС (Unix, Windows та ін.)?
   1) запобігання тупику;
   2) ігнорування проблеми в цілому;
   3) виявлення тупика;
   4) відновлення після тупика.
5. Один із способів боротьби з тупиками – скласти список усіх ресурсів і задовольняти запити процесів у порядку зростання номерів ресурсів. Яку з 4 умов виникнення тупика можна порушити таким чином?
   1) умову кругового (циклічного) очікування;
   2) умову невивантажуваності (відсутності перерозподілу) ресурсів;
   3) умову утримання і очікування ресурсів;
   4) умову взаємного виключення.
6. У системі з трьома процесами є 11 ресурсів, а потреба процесів в ресурсах описується таблицею:

Процес	Максимальна потреба в ресурсах	Виділені ресурси
P0	8	5
P1	11	3
P2	3	2

Стан системи є надійним (безпечним) чи ненадійним?

1) надійним;
2) ненадійним;
3) буде надійним, якщо максимальну потребу процесу P1 в ресурсах понизити до 10.

7. Потреба потоку відразу в декількох ресурсах є необхідною умовою:
   1) виникнення тупика;
   2) усунення блокування;
   3) очікування ресурсів;
   4) усунення тупика.
8. Чому алгоритм банкіра не підходить для систем, що підтримують «гарячу» заміну пристроїв? Тому що:
   1) алгоритм банкіра не може розпізнати новий пристрій в процесі роботи;
   2) йому необхідно перебудувати усі ресурсні структури, а це займає багато часу;
   3) для того щоб підключити новий пристрій, йому необхідно перервати свою роботу і звернутися до ОС для підключення нового ресурсу;
   4) алгоритм банкіра призначений для роботи тільки з постійним числом системних ресурсів.
9. Чи правда, що процеси не можуть опинитися в ситуації взаємного блокування в результаті звичайної конкурентної боротьби за процесорний час?
   1) ні, процесор – це ресурс, що розділяється, тому процеси можуть опинитися в ситуації взаємного блокування в результаті звичайної конкурентної боротьби за цей ресурс;
   2) так, оскільки ОС завжди може забрати цей ресурс у результаті закінчення кванта часу або призупинення процесу і виділити його іншому процесу, а потім повернути його назад;
   3) так, оскільки процесор в якийсь момент часу виділяється тільки одному процесу до кінця його роботи, то взаємного блокування ніколи не буде.
10. Яку з 4-х основних умов виникнення взаємного блокування буде порушено в разі, якщо користувач зможе видалити «зависле» завдання?
    1) умову взаємного виключення – кожен ресурс або виділений в даний момент тільки одному процесу, або доступний;
    2) умову утримання і очікування ресурсів – процес може утримувати виділені ресурси під час запиту (очікування) інших ресурсів;
    3) умову невивантажуваності (відсутності перерозподілу) – у процесу не можна примусово забрати раніше отримані ресурси. Процес, що володіє ними, повинен сам звільнити ресурси;
    4) умову циклічного очікування – існує замкнутий ланцюг процесів, кожен з яких утримує як мінімум один ресурс, необхідний процесу, наступному в ланцюзі після даного.

11. У системі з трьома процесами є 8 ресурсів, а потреба процесів в ресурсах описується наступною таблицею:

Процес	Максимальна потреба	У використанні ресурсів	Вимагається ресурсів
P_0	8	5	3
P_1	4	1	3
P_2	2	1	1

Стан системи є надійним (безпечним) чи ненадійним?
1) надійним;
2) ненадійним.

12. Як правильно боротися з тупиком, який може виникнути при використанні принтера?
1) ігнорувати проблему;
2) захистити принтер семафором;
3) організувати спулінг.

13. Яка з операційних систем більше схильна до тупиків?
1) система пакетної обробки;
2) система жорсткого реального часу;
3) система з розподілом часу.

14. Нехай є 5 процесів – *P0*, …, *P4*, і 3 типи ресурсів – ресурс *A* (7 примірників), ресурс *B* (2 примірники) і ресурс *C* (6 примірників). Нехай стан системи в момент $\mathbf{T}_0$ такий:

	Allocation			Request		
	A	**B**	**C**	**A**	**B**	**C**
$\mathbf{P_0}$	0	1	0	0	0	0
$\mathbf{P_1}$	2	0	0	2	0	2
$\mathbf{P_2}$	3	0	3	0	0	0
$\mathbf{P_3}$	2	1	1	1	0	0
$\mathbf{P_4}$	0	0	2	0	0	2

У цьому стані системи послідовність процесів <*P0, P2, P3, P1, P4*>:
1) безпечна;
2) небезпечна.

10 УПРАВЛІННЯ ПАМ'ЯТТЮ

Пам'ять є найважливішим ресурсом, який вимагає ретельного управління з боку мультипрограмної операційної системи. Розподілу підлягає вся оперативна пам'ять, що не зайнята операційною системою. Зазвичай ОС розташовується в наймолодших адресах, проте може займати і найстарші адреси. Функціями ОС з управління пам'яттю є:

– відстежування вільної і зайнятої пам'яті;
– виділення пам'яті процесам і звільнення пам'яті при завершенні процесів;
– витіснення процесів з оперативної пам'яті на диск, коли розміри основної пам'яті не достатні для розміщення в ній усіх процесів, і повернення їх в оперативну пам'ять, коли в ній звільняється місце;
– налаштування адрес програми на конкретну область фізичної пам'яті.

10.1 Основні поняття і вимоги до управління пам'яттю

10.1.1 Переміщення

У багатозадачній системі доступна основна пам'ять розподіляється між багатьма процесами. Програміст не знає, які програми будуть резидентно знаходитися в основній пам'яті під час виконання його програми. Крім того, для максимального завантаження процесора бажано мати великий пул (буфер) готових до виконання процесів. Для цього необхідно завантажувати і вивантажувати активні процеси з основної пам'яті. Вимога, щоб вивантажена з пам'яті програма була знову завантажена в одне і те ж місце, де вона була раніше, було б великим обмеженням. Тому вкрай бажано, щоб програма могла бути переміщена в іншу ділянку пам'яті.

Таким чином, заздалегідь невідомо, де саме буде розміщена програма. Крім того, програма може бути переміщена з однієї області пам'яті в іншу в разі закінчення іншої програми. Очевидно, що ОС необхідно знати місце розташування управляючої інформації, а також точки входу для початку виконання процесу. Оскільки управлінням пам'яттю займається ОС, вона ж розміщує процес в основній пам'яті, то вона автоматично отримує відповідні адреси.

10.1.2 Захист

Кожен процес має бути захищений від випадкового або з якимсь наміром небажаного впливу інших процесів. Тому код інших процесів не повинен мати можливості без дозволу звертатися до пам'яті цього процесу. Під час роботи програми необхідно виконати перевірку всіх звернень до пам'яті, які генеруються процесом, щоб переконатися, що всі вони звертаються тільки до пам'яті, яка виділена цьому процесу. Механізми підтримки переміщень забезпечують і підтримку захисту. Вимоги щодо захисту пам'яті повинні виконуватися на рівні процесора (апаратно), а не на рівні ОС.

10.1.3 Спільне використання

Будь-який механізм захисту повинен мати достатню гнучкість, для того щоб забезпечити можливість декільком процесам звертатися до однієї і тієї ж області основної пам'яті. Наприклад, якщо декілька процесів виконують один і той же машинний код, то буде вигідно дозволити кожному процесу працювати з однією і тією ж копією цього коду, а не створювати свою власну копію. Процесам, які взаємодіють при виконанні деякого завдання, може знадобитися загальний доступ до одних і тих же структур даних. Система управління пам'яттю повинна забезпечувати керований доступ до різних ділянок пам'яті.

10.1.4 Типи адрес

Для ідентифікації змінних і команд використовуються ***символьні імена*** (мітки), ***віртуальні адреси*** і ***фізичні адреси*** (рис. 10.1).

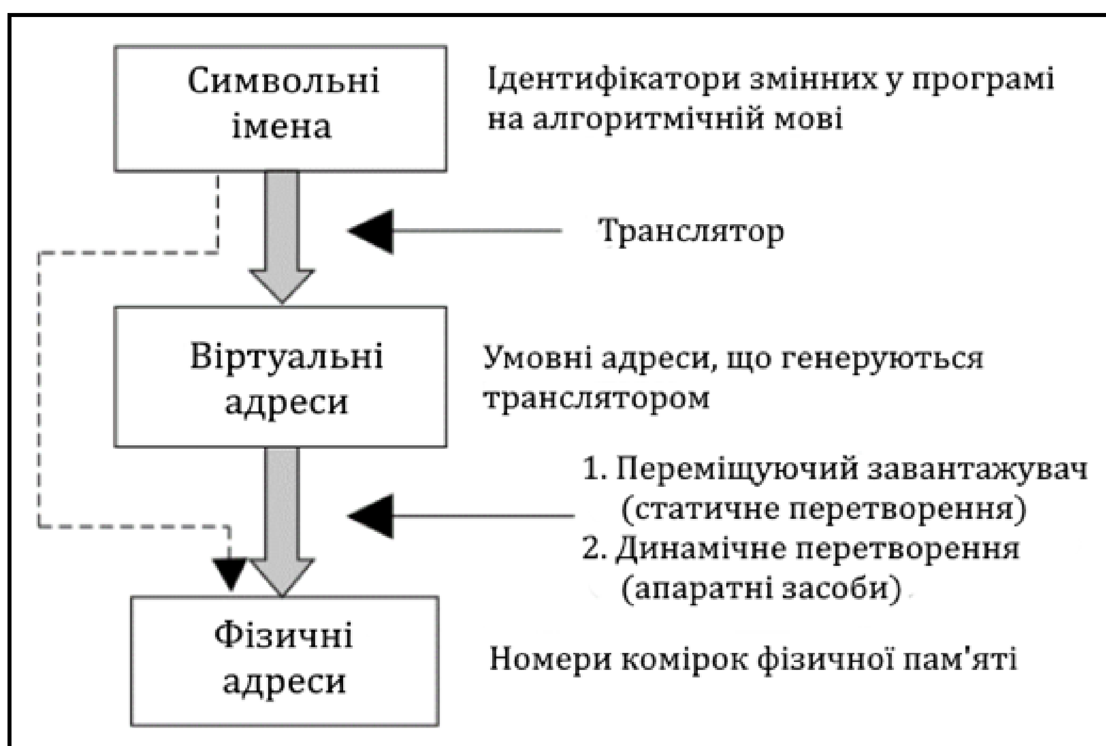


Рисунок 10.1 – Типи адрес

Символьні імена привласнює користувач при написанні програми алгоритмічною мовою або на асемблері.

Віртуальні адреси виробляє транслятор, який перекладає програму на машинну мову. Оскільки під час трансляції, в загальному випадку, невідомо в яке місце оперативної пам'яті буде завантажена програма, то транслятор привласнює змінним і командам віртуальні (умовні) адреси, вважаючи за умовчанням, що програма буде розміщена, починаючи з нульової адреси. Є ще поняття ***відносної адреси,*** що є окремим випадком віртуальної адреси (іноді – логічної адреси), коли адреса визначається положенням відносно деякої відомої точки (початку програми).

Сукупність віртуальних адрес процесу називається ***віртуальним адресним простором.*** Кожен процес має власний віртуальний адресний простір. Максимальний розмір віртуального адресного простору обмежується розрядністю адреси, властивій цій архітектурі комп'ютера, і, як правило, не співпадає з об'ємом фізичної пам'яті, наявним в комп'ютері. Наприклад, при використанні 32-розрядних віртуальних адрес цей діапазон задається межами 00000000_{16} і $FFFFFFFF_{16}$.

У різних операційних системах використовуються різні способи структуризації віртуального адресного простору. В одних ОС віртуальний адресний простір процесу подібно до фізичної пам'яті представлений у вигляді безперервної лінійної послідовності віртуальних адрес. Таку структуру адресного простору називають також *плоскою* (flat). При цьому віртуальною адресою є єдине число, яке представляє собою зміщення відносно початку віртуального адресного простору (звичайне це значення 000...000). Адресу такого типу називають ***лінійною віртуальною адресою***. Недоліки такої системи: одна ділянка може повністю заповнитися, але при цьому залишаться вільні ділянки. Можна звичайно переміщати ділянки, але це дуже складно.

Ці проблеми можна вирішити, якщо дати кожній ділянці незалежний адресний простір. Тому в багатьох ОС віртуальний адресний простір ділиться на частини, які називаються ***сегментами*** (або *секціями,* або *областями,* або іншими термінами). У цьому випадку окрім лінійної адреси може бути використана віртуальна адреса, що є парою чисел *(n, m),* де *n* визначає сегмент, а *m* – зміщення усередині сегменту.

Фізичні адреси (абсолютні, реальні) відповідають номерам елементів оперативної пам'яті, де насправді розташовані або будуть розташовані змінні і команди. Перехід від віртуальних адрес до фізичних може здійснюватися двома способами.

При застосуванні першого випадку заміну віртуальних адрес на фізичні робить спеціальна системна програма – ***переміщаючий завантажувач***. Переміщаючий завантажувач на підставі наявних у нього даних про початкову адресу фізичної пам'яті, в яку належить завантажувати програму, і інформації, наданої транслятором про адресно-залежні константи програми, виконує завантаження програми, поєднуючи її із заміною віртуальних адрес на фізичні.

Другий спосіб полягає в тому, що програма завантажується в пам'ять у незміненому виді у віртуальних адресах, при цьому операційна система фіксує зміщення дійсного розташування програмного коду відносно віртуального адресного простору.

Під час виконання програми при кожному зверненні до оперативної пам'яті виконується перетворення віртуальної адреси у фізичну. Другий спосіб є гнучкішим, він допускає переміщення програми під час її виконання, тоді як переміщаючий завантажувач жорстко прив'язує програму до спочатку виділеної їй ділянки пам'яті. У той же час використання переміщаючого завантажувача зменшує накладні витрати, оскільки перетворення кожної віртуальної адреси відбувається тільки один раз під час завантаження, а в другому випадку – кожного разу при зверненні за цією адресою.

В деяких випадках (зазвичай в спеціалізованих системах), коли заздалегідь точно відомо, в якій області оперативної пам'яті виконуватиметься програма, транслятор видає виконуваний код відразу у фізичних адресах.

А тепер розглянемо алгоритми розподілу пам'яті, які застосовуються і застосовувалися в різних ОС.

Усі методи управління пам'яттю можуть бути розділені на два класи: методи, які використовують переміщення процесів між оперативною пам'яттю і диском, і методи, які не роблять цього (рис. 10.2).

Рисунок 10.2 – Класифікація методів розподілу пам'яті

Чи слід призначати кожному процесу одну безперервну область фізичної пам'яті або можна виділяти пам'ять «ділянками»? Чи повинні сегменти програми, завантажені в пам'ять, знаходитися на одному місці впродовж усього періоду виконання процесу або можна їх час від часу переміщати? Що робити, якщо сегменти програми не поміщаються в наявну пам'ять?

Різні ОС по-різному відповідають на ці і інші базові питання управління пам'яттю. Далі будуть розглянуті найзагальніші підходи до розподілу пам'яті, які були характерні для різних періодів розвитку операційних систем. Деякі з них зберегли актуальність і широко використовуються в сучасних ОС, інші ж представляють в основному тільки пізнавальний інтерес, хоча їх і сьогодні можна зустріти в спеціалізованих системах.

10.2. Розподіл пам'яті фіксованими розділами

Найпростішим способом управління оперативною пам'яттю без використання дискового простору є розділення її на області з фіксованими межами. Це може бути виконано під час генерації ОС. У тому випадку, коли розділи мають однаковий розмір, розміщення процесів в пам'яті є тривіальною

задачою. Не має значення, в якому з вільних розділів буде розміщений процес. Але при цьому підході є і свої труднощі:

1. Програма може бути занадто велика для розміщення в розділі. В цьому випадку програміст повинен розробити програму, що використовує ***оверлеї*** (перекриття).
2. Використання пам'яті при цьому вкрай неефективно. Так, маленька програма все одно займатиме цілком увесь розділ, при цьому залишається невикористана пам'ять. Цей феномен невикористаної пам'яті називається ***внутрішньою фрагментацією***.

Коли розділи мають різні розміри, чергова задача, що поступило на виконання, поміщається або в загальну чергу (рис. 10.3, *а*), або в чергу до деякого розділу (рис. 10.3, *б*).

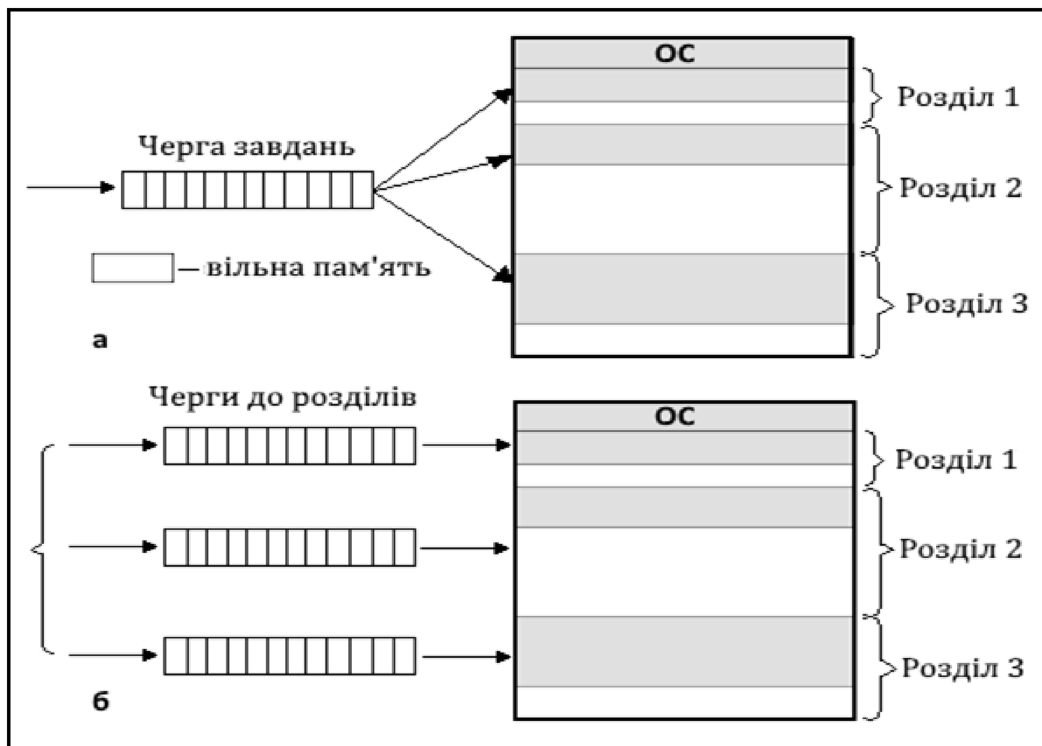


Рисунок 10.3 – Розподіл пам'яті фіксованими розділами:
а) – із загальною чергою; б) – з окремими чергами для кожного розділу

Підсистема управління пам'яттю в цьому випадку виконує такі задачі:

– вибирає відповідний розділ, порівнюючи розмір програми, що поступила на виконання, і вільних розділів;
– здійснює завантаження програми і налаштування адрес.

Перевага такого підходу полягає в тому, що процеси можуть бути розподілені між розділами пам'яті так, щоб мінімізувати внутрішню фрагментацію. Використання розділів різного розміру в порівнянні з використанням розділів однакового розміру надає додаткову гнучкість цьому методу.

При очевидній перевазі (простоті реалізації) цей метод, у цілому, має істотний недолік – жорсткість. Оскільки в кожному розділі може виконуватися

тільки одна програма, то рівень мультипрограмування заздалегідь обмежений числом розділів, не залежно від того, який розмір мають програми.

Навіть якщо програма має невеликий об'єм, вона займатиме увесь розділ, що призводить до неефективного використання пам'яті. З іншого боку, навіть якщо об'єм оперативної пам'яті машини дозволяє виконати деяку програму, розбиття пам'яті на розділи не дозволяє зробити цього.

Оскільки модель з фіксованими розділами обмежує віртуальний адресний простір розмірами реальної пам'яті, виникає проблема великих програм, що не вміщуються в доступну пам'ять. Подолання цього обмеження досягається створенням програм з оверлейною структурою (overlay – перекриття).

Структура міжмодульних викликів оверлейної програми має вигляд дерева. В корені дерева (нульовий рівень) знаходиться ***модуль-монітор***, з якого здійснюється звернення до модулів, розташованих на гілках дерева. Кожен модуль першого рівня у свою чергу може бути коренем піддерева і так далі. Принцип оверлеїв або перекриття полягає в тому, що гілки дерева займають одні і ті ж області у віртуальному адресному просторі програми.

Оскільки модулі, розташовані в різних гілках дерева, не можуть виконуватися одночасно, то тільки монітор є резидентним в оперативній пам'яті весь час виконання програми, гілки ж змінюють одина одну в одній і тій же області пам'яті. Насправді побудувати програму, що має ідеальну деревовидну структуру, досить важко.

У системах, що підтримують розвинені засоби створення оверлейних програм (OS/360), велика робота покладається на редактор зв'язків, який формує вміст адресного простору процесу відповідно до оверлейної структури, що описується програмістом за допомогою спеціальної мови.

У цілому можна відмітити, що схеми з фіксованими розділами відносно прості, тому пред'являються мінімальні вимоги до ОС; накладні витрати роботи процесора на розподіл пам'яті невеликі. Проте у цих схем є серйозні недоліки.

1. Кількість розділів, визначена в момент генерації системи, обмежує кількість активних процесів.
2. Оскільки розміри розділів встановлюються заздалегідь під час генерації системи, невеликі завдання призводять до неефективного використання пам'яті.

Фіксований розподіл нині практично не використовується. Прикладом ОС з використанням цієї технології може служити рання ОС IBM для мейнфреймів OS/MFT (Multiprogramming with a Fixed number of Tasks – багатозадачна з фіксованою кількістю задач).

10.3 Динамічний розподіл пам'яті

Для подолання складнощів, пов'язаних з фіксованим розподілом, був розроблений альтернативний підхід, відомий як динамічний розподіл.

При динамічному розподілі пам'яті без використання дискового простору пам'ять машини не ділиться заздалегідь на розділи. Спочатку вся пам'ять вільна. Кожній новій задачі виділяється необхідна їй пам'ять. Якщо достатній об'єм

пам'яті відсутній, то задача не приймається на виконання і стоїть в черзі. Після завершення задачі пам'ять звільняється, і на це місце може бути завантажене інша задача. Таким чином, в довільний момент часу оперативна пам'ять є випадковою послідовністю зайнятих і вільних ділянок (розділів) довільного розміру. На рис. 10.4 показаний стан пам'яті в різні моменти часу при використанні динамічного розподілу.

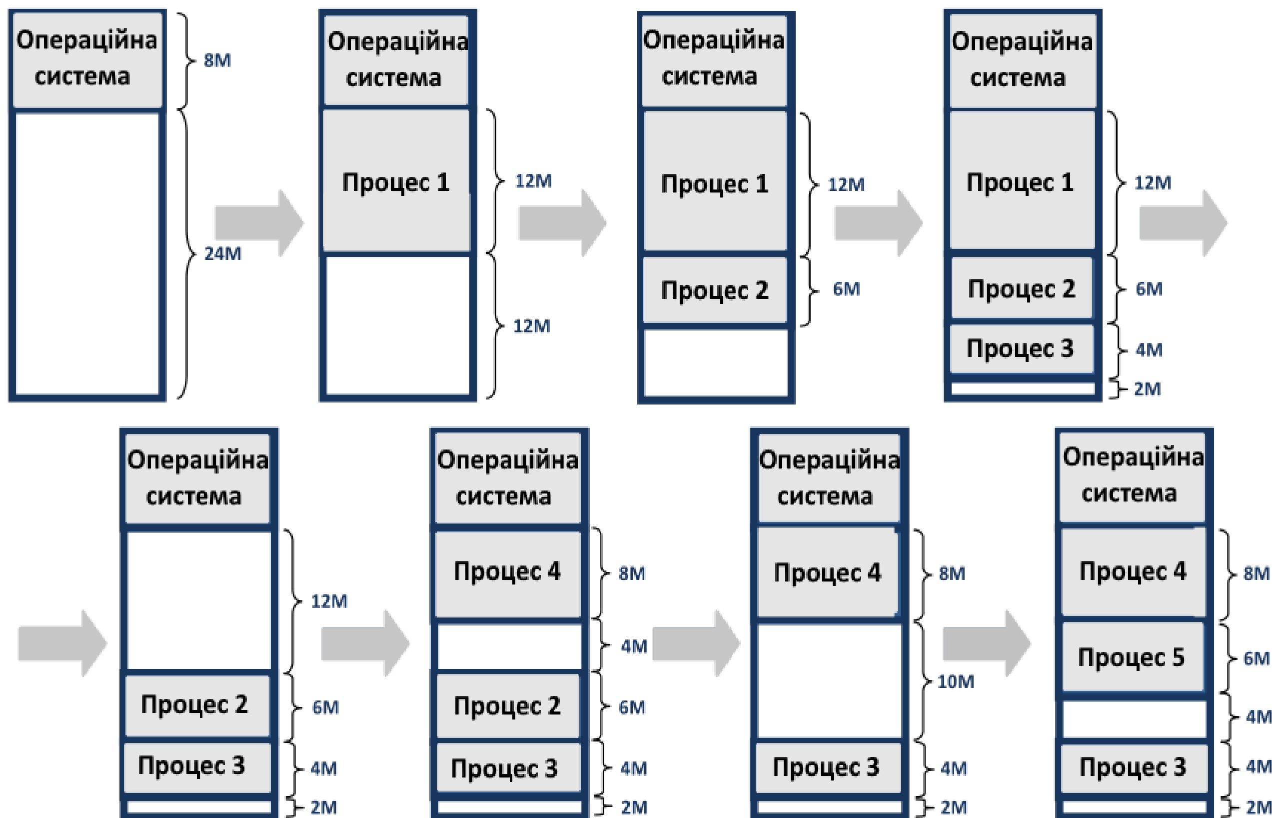


Рисунок 10.4 – Розподіл 32 Мб пам'яті динамічними розділами

Так, у момент t_0 в пам'яті знаходиться тільки ОС, а до моменту t_1 пам'ять розділена між 5 задачами, причому задача *П4,* завершуючись, покидає пам'ять. На звільнене місце після задачі *П4* завантажується задача *П6,* що поступила в момент t_3.

Задачами ОС при реалізації цього методу управління пам'яттю є:

- ведення таблиць вільних і зайнятих областей, в яких указуються початкові адреси і розміри ділянок пам'яті;
- при надходженні нової задачі ОС аналізує запит, переглядає таблиці вільних областей і вибирає розділ, розмір якого достатній для розміщення нової задачі;
- завантаження задачі у виділений їй розділ і коригування таблиць вільних і зайнятих областей;
- після завершення задачі ОС коригує таблиці вільних і зайнятих областей.

Програмний код не переміщається під час виконання, тобто може бути проведене одноразове налаштування адрес за допомогою використання переміщаючого завантажувача.

Вибір розділу для нової задачі може здійснюватися за різними правилами, такими, наприклад, як «перший розділ достатнього розміру», що попався, або «розділ, що має найменший достатній розмір». Усі ці правила мають свої переваги і недоліки.

У порівнянні з методом розподілу пам'яті фіксованими розділами цей метод має набагато більшу гнучкість, але йому властивий дуже серйозний недолік – ***зовнішня фрагментація пам'яті***. Зовнішня фрагментація – це наявність великого числа несуміжних ділянок вільної пам'яті дуже маленького розміру (фрагментів). Настільки маленького, що жодна з програм, яка знову поступає, не може поміститися ні в одній з ділянок, хоча сумарний об'єм фрагментів може скласти значну величину, що набагато перевищує необхідний об'єм пам'яті.

Свого часу динамічний розподіл використала ОС IBM для мейнфреймів OS/MVT (Multiprogramming with a Variable number of Tasks – багатозадачна зі змінною кількістю задач). Пізніше цей же підхід до розподілу пам'яті був використаний в ОС ЄС ЕОМ.

10.4 Переміщувані розділи

Одним з методів боротьби із зовнішньою фрагментацією є переміщення усіх зайнятих ділянок у бік старших або у бік молодших адрес, так, щоб уся вільна пам'ять утворювала єдину вільну область (рис. 10.5).

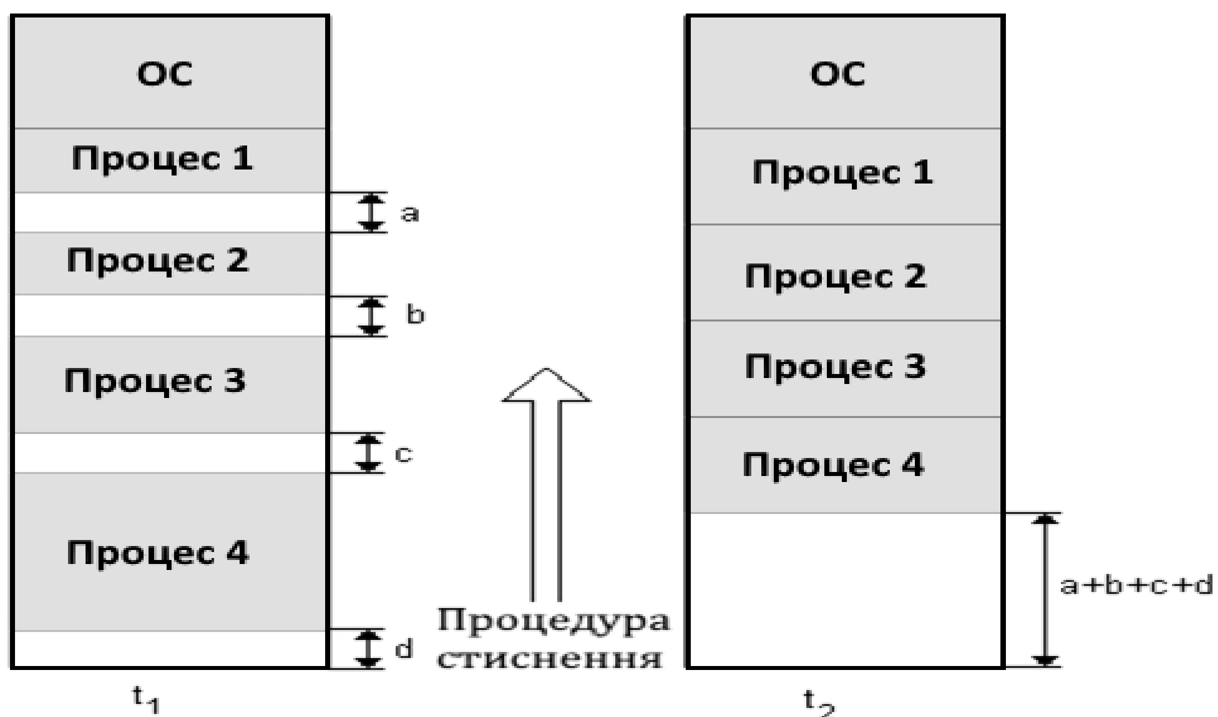


Рисунок 10.5 – Розподіл пам'яті переміщуваними розділами

На додаток до функцій, які виконує ОС при розподілі пам'яті змінними розділами, в даному випадку вона повинна ще час від часу копіювати вміст розділів з одного місця пам'яті в інше, коригуючи таблиці вільних і зайнятих областей. Ця процедура називається «ущільненням» або «стискуванням».

Перелічимо функції операційної системи з управління пам'яттю в цьому випадку:

1. Переміщення всіх зайнятих ділянок у бік старших або молодших адрес при кожному завершенні процесу або для новоствореного процесу в разі відсутності розділу достатнього розміру.
2. Корекція таблиць вільних і зайнятих областей.
3. Зміна адрес команд і даних, до яких звертаються процеси при їх переміщенні в пам'яті за рахунок використання відносної адресації.
4. Апаратна підтримка процесу динамічного перетворення відносних адрес в абсолютні адреси основної пам'яті.
5. Захист пам'яті, що виділяється процесу, від взаємного впливу інших процесів.

Стискування (ущільнення) може виконуватися або при кожному завершенні задачі, або тільки тоді, коли для задачі, яка знову поступила, немає вільного розділу достатнього розміру. У першому випадку потрібно менше обчислювальної роботи при коригуванні таблиць, а в другому – рідше виконується процедура стискування. Оскільки програми переміщаються по оперативній пам'яті в ході свого виконання, то перетворення адрес з віртуальної форми у фізичну повинне виконуватися динамічним способом.

Перевагами розподілу пам'яті переміщуваними розділами є ефективне використання оперативної пам'яті, виключення внутрішньої і зовнішньої фрагментації.

Хоча процедура стискування і призводить до ефективнішого використання пам'яті, вона може зажадати значного часу, що часто знижує переваги цього методу.

Ситуація ускладнюється, якщо процеси по ходу роботи можуть займати різне місце розташування в розділах. У цих випадках розташування команд і даних, до яких звертається процес, не є фіксованим і змінюється всякий раз при вивантаженні, завантаженні або переміщенні процесу. Для вирішення цієї проблеми в програмах використовуються ***відносні адреси***. Це означає, що всі посилання на пам'ять у завантажуваному процесі даються відносно початку цієї програми. Таким чином, для коректної роботи програми потрібен апаратний механізм, який би транслював відносні адреси у фізичні в процесі виконання команди, яка звертається до пам'яті. Один із способів трансляції показаний на рис. 10.6.

Коли процес переходить у стан виконання, то в спеціальний базовий регістр процесу завантажується початкова адреса процесу в основній пам'яті. Крім того, використовується «граничний» (bounds) регістр, в якому міститься адреса останньої комірки програми. Ці значення заносяться в регістри при завантаженні програми в основну пам'ять.

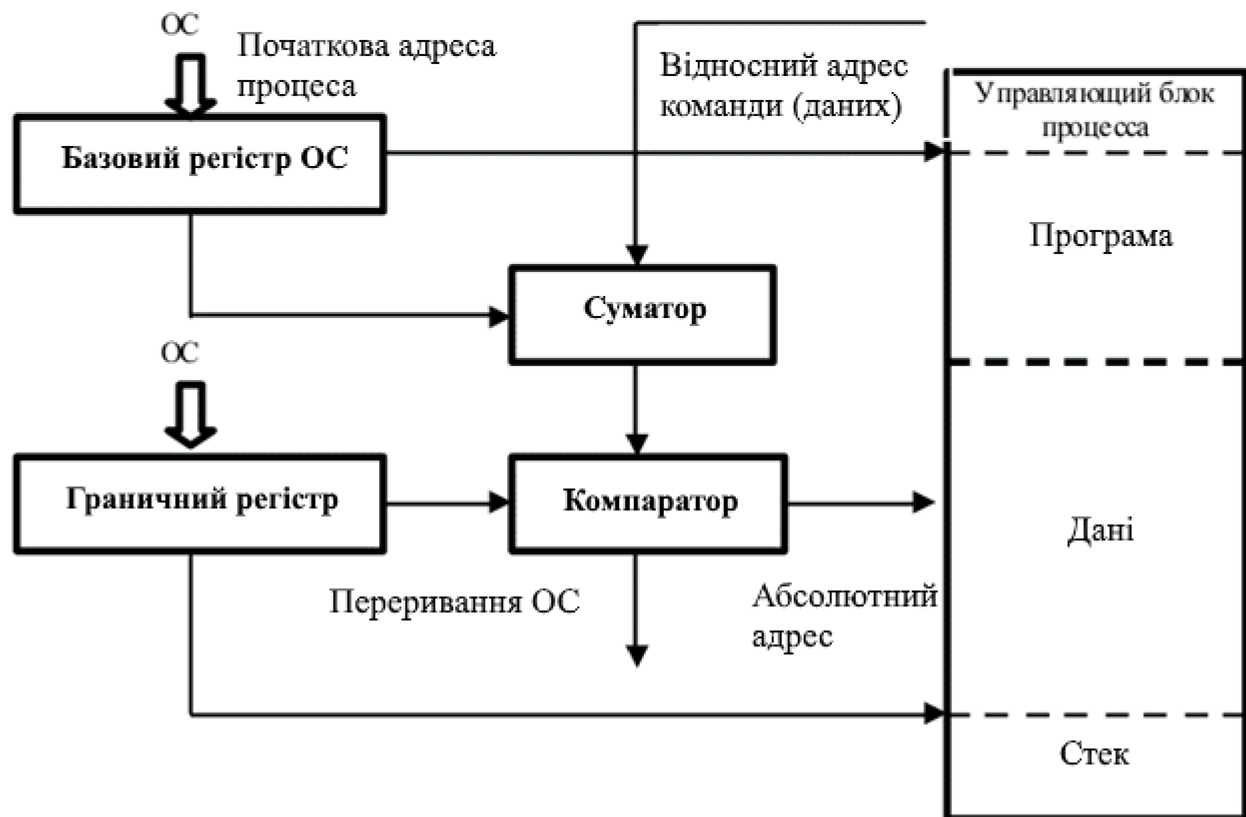


Рисунок 10.6 – Перетворення адрес

При виконанні процесу відносні адреси в командах обробляються процесором в два етапи. Спочатку до відносної адреси додається значення базового регістра для отримання абсолютної адреси. Потім отримана абсолютна адреса порівнюється зі значенням в граничному регістрі. Якщо отримана абсолютна адреса належить цьому процесу, то команда може бути виконана. Інакше генерується переривання, що відповідає цій помилці.

10.5 Система двійників

Як фіксований, так і динамічний розподіл пам'яті мають переваги і недоліки. Компромісним у цьому плані є система двійників, в якій пам'ять розподіляється на блоки розміром $\mathbf{2^K}$, $\mathbf{L <= K <= U}$, де: – $\mathbf{2^L}$ – мінімальний розмір блоку пам'яті; $\mathbf{2^U}$ – найбільший розмір блоку пам'яті (спочатку вся доступна пам'ять).

При запиті пам'яті розміром **S,** таким, що $\mathbf{2^{U-1} < S <= 2^U}$, виділяється увесь блок. Інакше блок розділяється на два рівних двійники з розмірами $\mathbf{2^{U-1}}$. Якщо $\mathbf{2^{U-2} < S <= 2^{U-1}}$, то за запитом виділяється один з двох двійників; інакше один з двійників знову ділиться навпіл.

Цей процес триває до тих пір, поки не буде згенерований найменший блок, розмір якого не менше **S**. Система двійників постійно веде список «дір» (доступних блоків) для кожного розміру $\mathbf{2^i}$. Діра може бути видалена зі списку $(i+1)$ розділенням її навпіл і внесенням двох нових дір розміру $\mathbf{2^i}$ в список i. Коли

пара двійників в списку *i* виявляється звільненою, вони видаляються зі списку і об'єднуються в єдиний блок в списку *i+1*.

Нижче наведено приклад (рис. 10.7) використання блоку з початковим розміром 1 Мб. Перший запит *A* – на 100 Кб (для нього потрібен блок розміром 128 Кб). Для цього початковий блок ділиться на два двійники по 512 Кб. Потім перший з них ділиться на два двійники розміром 256 Кб, і, у свою чергу, перший двійник, який вийшов при цьому поділі, також ділиться навпіл (128 Кб). Один із двійників розміром 128 Кб виділяється процесу *A*. Наступний запит *B* вимагає 240 Кб. Такий блок є в наявності і виділяється. Процес триває з поділом і злиттям двійників при необхідності. Зверніть увагу, що після звільнення блоку *E* відбувається злиття двійників по 128 Кб в один блок розміром 256 Кб, який, у свою чергу, тут же зливається зі своїм двійником.

<table>
<tr><td>Початковий блок</td><td colspan="6">1Мб</td></tr>
<tr><td>Запит A=100K</td><td>A128</td><td colspan="2">128K</td><td>256K</td><td colspan="2">512K</td></tr>
<tr><td>Запит B=240K</td><td>A128</td><td colspan="2">128K</td><td>B256</td><td colspan="2">512K</td></tr>
<tr><td>Запит C=63K</td><td>A128</td><td>C64</td><td>64K</td><td>B256</td><td colspan="2">512K</td></tr>
<tr><td>Запит D=256K</td><td>A128</td><td>C64</td><td>64K</td><td>B256</td><td>D256</td><td>256K</td></tr>
<tr><td>Звільнення B</td><td>A128</td><td>C64</td><td>64K</td><td>256K</td><td>D256</td><td>256K</td></tr>
<tr><td>Звільнення A</td><td>128K</td><td>C64</td><td>64K</td><td>256K</td><td>D256</td><td>256K</td></tr>
<tr><td>Запит E=75K</td><td>E128</td><td>C64</td><td>64K</td><td>256K</td><td>D256</td><td>256K</td></tr>
<tr><td>Звільнення C</td><td>E128</td><td colspan="2">128K</td><td>256K</td><td>D256</td><td>256K</td></tr>
<tr><td>Звільнення E</td><td colspan="4">512K</td><td>D256</td><td>256K</td></tr>
<tr><td>Звільнення D</td><td colspan="6">1Мб</td></tr>
</table>

Рисунок 10.7 – Приклад роботи системи двійників

Система двійників є розумним компромісом для подолання недоліків схем фіксованого і динамічного розподілу, але в сучасних операційних системах її перевершує віртуальна пам'ять, заснована на сторінковій організації і сегментації. Однак система двійників знайшла застосування в сучасних ОС як ефективний засіб розподілу і звільнення паралельних програм. Модифікована версія системи двійників використовується для розподілу пам'яті ядром UNIX.

10.6 Поняття віртуальної пам'яті

У наступних розділах мова піде про найпоширенішу нині схему управління пам'яттю, відомою як віртуальна пам'ять, у рамках якої здійснюється складний зв'язок між апаратним і програмним забезпеченням. Розбиття адресного простору процесу на частини і динамічна трансляція адреси дозволили виконувати процес навіть за відсутності деяких його компонентів в оперативній пам'яті. Наслідком такої стратегії є можливість виконання великих програм, розмір яких може перевищувати розмір оперативної пам'яті.

Вже досить давно користувачі зіштовхнулися з проблемою розміщення в пам'яті програм, розмір яких перевищував вільну пам'ять, що є в наявності. Рішенням було розбиття програми на частини, що називаються ***оверлеями***. 0-ий

оверлей починав виконуватися першим. Коли він закінчував своє виконання, він викликав інший оверлей. Усі оверлеї зберігалися на диску і переміщалися між пам'яттю і диском засобами операційної системи. Проте розбиття програми на частини і планування їх завантаження в оперативну пам'ять повинен був здійснювати програміст.

Розвиток методів організації обчислювального процесу в цьому напрямі привів до появи методу, відомого під назвою ***віртуальна пам'ять***. Віртуальним називається ресурс, який користувачеві або призначеній для користувача програмі представляється таким, що має властивості, якими він насправді не володіє. Так, наприклад, користувачеві може бути надана віртуальна оперативна пам'ять, розмір якої перевершує усю наявну в системі реальну оперативну пам'ять. Користувач пише програми так, як ніби в його розпорядженні є однорідна оперативна пам'ять великого об'єму, але насправді усі дані, використовувані програмою, зберігаються на одному або декількох різнорідних пристроях, зазвичай на дисках, і при необхідності частинами відображаються в реальну пам'ять.

Уперше віртуальна пам'ять була використана в операційній системі машини Atlas (Англія, 1962 рік) [12]. Незабаром після цього віртуальна пам'ять стала широко використовуватися в комерційних системах. Atlas Supervisor була здатна виконувати до 16 задач одночасно, встановивши новий стандарт продуктивності.

Спираючись на емпіричні (засновані на спостереженнях) результати, можна відмітити, що основний об'єм пам'яті, яку займає процес, велику частину часу залишається вільним, то існує таке правило «***дев'яносто до десяти»,*** або ***правило локалізації (локальності),*** яке стверджує, що ***90% звернень до пам'яті в процесі доводиться на 10% його адресного простору***.

Локальність – емпірична закономірність, що зв'язує близькі за часом або в просторі події. Якщо у вашому місті сонячно, то, ймовірно, але не напевно, поблизу від нього теж сонячно. Якщо зараз в місті гарна погода, то, ймовірно, але не напевно, вона була хорошою деякий час назад і залишиться хорошою ще деякий час у майбутньому.

У застосуванні до послідовностей звернень до пам'яті ***просторова локальність*** означає, що процес, який звертався раніше за деякою адресою, ймовірно, звертатиметься і до близьких до нього адрес. ***Часова локальність*** означає, що процес, який звертався нещодавно за якоюсь адресою, ймовірно, звернеться до неї знову.

Застосовуючи модель локальності, в основній пам'яті можна зберігати постійно тільки ті розділи адресного простору, які дійсно використовуються в конкретний момент. При цьому невикористані розділи адресного простору можна ставити у відповідність менш швидкій пам'яті, наприклад, простору на жорсткому диску, а в цей час інші процеси можуть використовувати основну пам'ять, займану раніше цими розділами. Якщо ж ці розділи знадобляться знову, то вони можуть бути завантажені з диска в основну пам'ять (може бути в інший адресний простір).

Таким чином, ***віртуальна пам'ять*** – це сукупність програмно-апаратних засобів, що дозволяють користувачам писати програми, розмір яких перевищує наявну оперативну пам'ять. Для цього віртуальна пам'ять розв'язує такі задачі:

1. Розміщує дані в пристроях різного типу, наприклад, частина програми в оперативній пам'яті, а частина на диску.
2. Переміщає в міру необхідності дані між пристроями різного типу, наприклад, підвантажує потрібну частину програми з диска в оперативну пам'ять.
3. Перетворює віртуальні адреси у фізичні.

Усі ці дії виконуються автоматично, без участі програміста, тобто механізм віртуальної пам'яті є прозорим стосовно користувача. Найбільш поширеними реалізаціями віртуальної пам'яті є ***сторінковий***, ***сегментний*** і ***сторінково-сегментний*** розподіл пам'яті, а також ***свопінг***.

10.7 Сторінковий розподіл пам'яті

Як розділи з різними фіксованими розмірами, так і розділи змінного розміру недостатньо ефективно використовують оперативну пам'ять. Результатом роботи перших стає внутрішня фрагментація, результатом останніх – зовнішня.

При сторінковій організації віртуальний адресний простір кожного процесу ділиться на частини однакового, невеликого фіксованого для цієї системи розміру, що називаються ***віртуальними сторінками***. У загальному випадку розмір віртуального адресного простору не є кратним розміру сторінки, тому остання сторінка кожного процесу доповнюється фіктивною областю.

Уся оперативна пам'ять машини також ділиться на частини такого ж розміру, що називаються ***фізичними сторінками*** (або ***блоками)***. Вільні блоки оперативної пам'яті ще відомі як ***кадри*** (frames) або ***фрейми***. Кожен кадр може містити одну сторінку даних.

Розмір сторінки звичайно вибирається рівним степеню двійки: 1024, 2048, 4096 тощо. Це дозволяє спростити механізм перетворення адрес.

При створенні процесу частина його віртуальних сторінок (початкові сторінки кодового сегменту і сегменту даних) поміщається в оперативну пам'ять. Копія усього віртуального адресного простору процесу знаходиться на диску. Частина процесу, яка розташована в деякий момент часу в основній пам'яті, називається ***резидентною множиною*** процесу. Суміжні віртуальні сторінки не обов'язково розташовуються в суміжних фізичних сторінках.

Операційна система створює для кожного процесу інформаційну структуру – ***таблицю сторінок***. Кількість дескрипторів у таблиці і їх розмір підбираються такими, щоб об'єм таблиці виявився рівним об'єму сторінки. Адреса таблиці сторінок включається в контекст процесу. При активізації чергового процесу ОС завантажує адресу його таблиці сторінок в спеціальний регістр. У таблиці сторінок встановлюється відповідність між номерами віртуальних і фізичних сторінок (Nф), які завантажені в оперативну пам'ять, або робиться відмітка, що віртуальна сторінка вивантажена на диск (ВП, рис. 10.8).

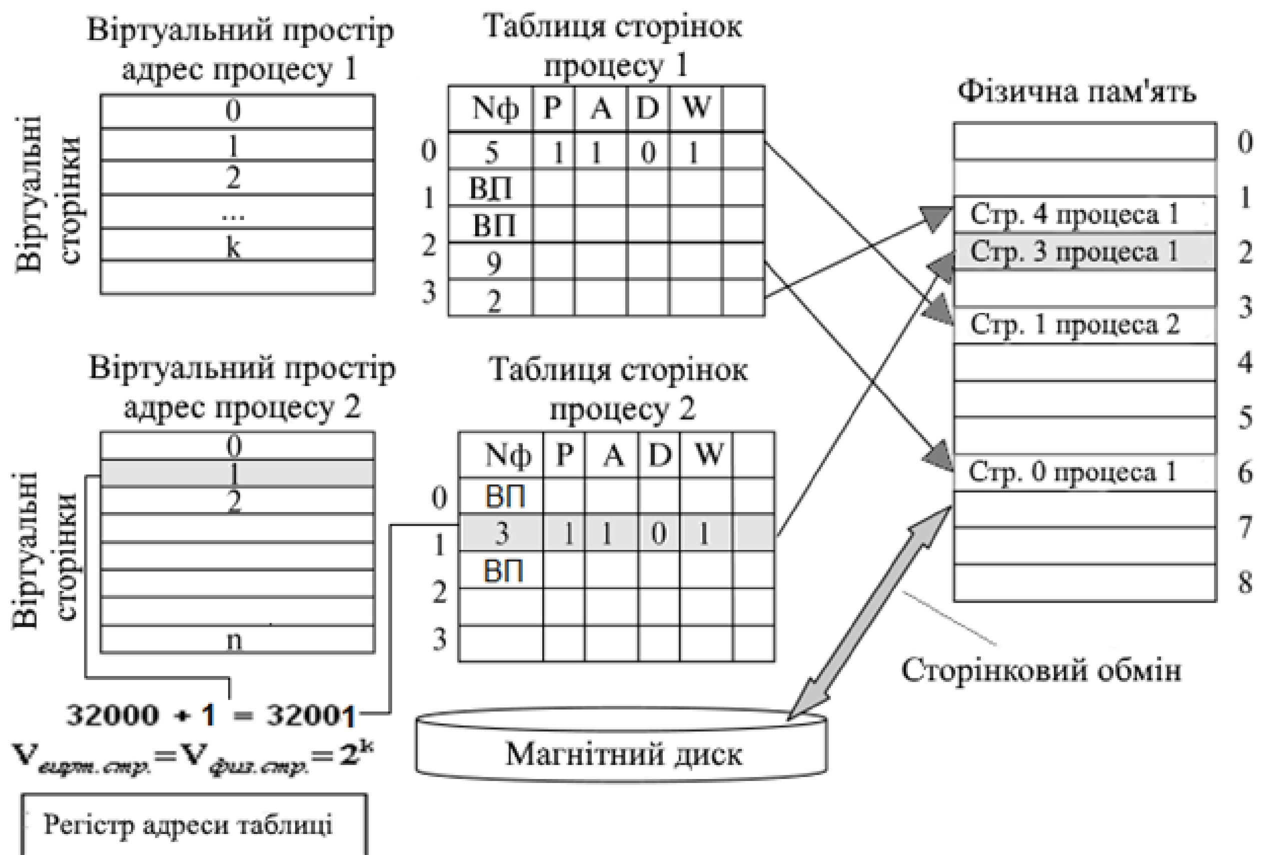


Рисунок 10.8 – Таблиці сторінок віртуальної пам'яті

Крім того, в кожному записі таблиці (дескриптор сторінки) міститься управляюча інформація:

– ознака присутності **P**, що встановлюється в одиницю, якщо ця сторінка знаходиться в оперативній пам'яті;
– ознака зміни (модифікації) сторінки **D**, яка встановлюється в одиницю всякий раз, коли робиться запис за адресою, що належить до цієї сторінки;
– ознака звернення **A** до сторінки, що називається також бітом доступу, який встановлюється в одиницю при кожному зверненні за адресою, що належить до цієї сторінки;
– ознака захисту сторінки **W** (читання, читання/запис);
– інші управляючі службові біти, наприклад, для цілей захисту або спільного використання пам'яті на рівні сторінок.

Ознаки присутності, модифікації і звернення в більшості моделей сучасних процесорів встановлюються апаратно, схемами процесора при виконанні операції з пам'яттю. Інформація з таблиць сторінок використовується для вирішення питання про необхідність переміщення тієї або іншої сторінки між пам'яттю і диском, а також для перетворення віртуальної адреси у фізичну. Самі таблиці сторінок, також як і описувані ними сторінки, розміщуються в оперативній пам'яті. Адреса таблиці сторінок включається в контекст відповідного процесу. При активізації чергового процесу операційна система завантажує адресу його таблиці сторінок в спеціальний регістр процесора.

При кожному зверненні до пам'яті відбувається читання з таблиці сторінок інформації про віртуальну сторінку, до якої сталося звернення. Якщо ця віртуальна сторінка знаходиться в оперативній пам'яті, то виконується перетворення віртуальної адреси у фізичну. Якщо ж потрібна віртуальна сторінка в даний момент вивантажена на диск, то відбувається так зване ***сторінкове переривання***. Процес, що виконується, переводиться в стан очікування, і активізується інший процес з черги готових.

Паралельно програма обробки сторінкового переривання знаходить на диску необхідну віртуальну сторінку і намагається завантажити її в оперативну пам'ять. Якщо в пам'яті є вільна фізична сторінка, то завантаження виконується негайно, якщо ж вільних сторінок немає, то вирішується питання, яку сторінку слід вивантажити з оперативної пам'яті.

У цій ситуації може бути використано багато різних критеріїв вибору, але найпопулярніші з них такі:

- сторінка, яка найдовше не використалася;
- перша, яка попалась, сторінка;
- сторінка, до якої останнім часом було менше всього звернень.

У деяких системах використовується поняття ***робочої множини сторінок***. Робоча множина визначається для кожного процесу і є переліком найчастіше використовуваних сторінок, які повинні постійно знаходитися в оперативній пам'яті і тому не підлягають вивантаженню.

Після того, як вибрана сторінка, яка повинна покинути оперативну пам'ять, аналізується її ознака модифікації (з таблиці сторінок). Якщо виштовхана сторінка з моменту завантаження була модифікована, то її нова версія має бути переписана на диск. Якщо ні, то вона може бути просто знищена, тобто відповідна фізична сторінка оголошується вільною. Розглянемо механізм перетворення віртуальної адреси у фізичну при сторінковій організації пам'яті.

Віртуальна адреса при сторінковому розподілі представлена у вигляді пари (P, S_V), де P – порядковий номер віртуальної сторінки процесу (нумерація сторінок починається з 0), а S_V – зміщення в межах віртуальної сторінки (I – індекс). Фізична адреса також може бути представлена у вигляді пари (N, S_F), де N – номер фізичної сторінки, а S_F – зміщення в межах фізичної сторінки. Задача підсистеми віртуальної пам'яті полягає у відображенні (P, S_V) в (N, S_F).

Перш ніж приступити до розгляду схеми перетворення віртуальної адреси у фізичну, зупинимося на двох базисних властивостях сторінкової організації.

Перша з них полягає в тому, що об'єм сторінки вибирається рівним степені двійки – 2^k. З цього виходить, що зміщення S може бути отримане простим відділенням k молодших розрядів в двійковому записі адреси, а старші розряди адреси, що залишилися, є двійковим записом номера сторінки. При цьому неважливо, є сторінка віртуальною або фізичною. Наприклад, якщо розмір сторінки 1 Кб (2^{10}), то з двійкового запису адреси $5071_8 = 101\ 000\ 111\ 001_2$ можна визначити, що він належить сторінці в двійковому виразі 10_2, і зміщений відносно її початку на $1\ 000\ 111\ 001_2$ байт.

Тобто, номер сторінки і її початкова адреса легко можуть бути отримані один з іншого доповненням або відкиданням *k* нулів, що відповідають зміщенню.

Саме з цієї причини часто говорять, що таблиця сторінок містить початкову фізичну адресу сторінки в пам'яті (а не номер фізичної сторінки), хоча насправді в таблиці вказані тільки старші розряди адреси. Початкова адреса сторінки називається базовою адресою. Друга властивість полягає в тому, що в межах сторінки безперервна послідовність віртуальних адрес однозначно відображається в безперервну послідовність фізичних адрес, тобто зміщення у віртуальній і фізичній адресах S_V і S_F рівні між собою ($S_V = S_F$).

Звідси слідує проста схема перетворення віртуальної адреси у фізичну (рис. 10.9). Молодші розряди фізичної адреси, що відповідають зміщенню, створюються перенесенням такої ж кількості молодших розрядів з віртуальної адреси. Старші розряди фізичної адреси, що відповідають номеру фізичної сторінки, визначаються з таблиці сторінок, в якій вказується відповідність віртуальних і фізичних сторінок.

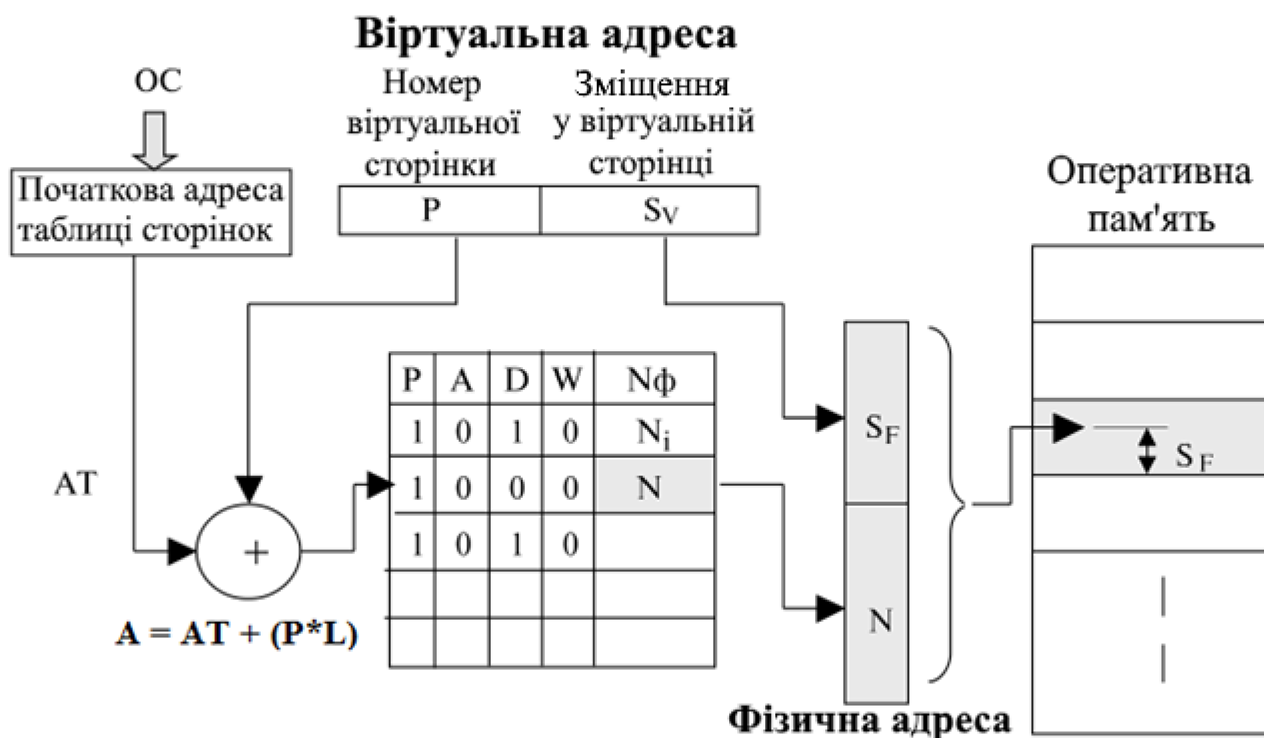


Рисунок 10.9 – Перетворення віртуальної адреси у фізичну при сторінковій організації пам'яті

При кожному зверненні до оперативної пам'яті апаратними засобами виконуються такі дії:

1. Із спеціального регістра процесора витягається адреса AT таблиці сторінок активного процесу. На підставі початкової адреси таблиці сторінок, номери віртуальної сторінки P (старші розряди віртуальної адреси) і довжини окремого запису в таблиці сторінок L (системна константа) визначається адреса потрібного дескриптора в таблиці сторінок: A=AT(P*L).
2. З цього дескриптора витягається номер відповідної фізичної сторінки – N.
3. До номера фізичної сторінки приєднується зміщення S_V (молодші розряди віртуальної адреси).

Саме для зменшення часу перетворення адрес в усіх процесорах передбачений апаратний механізм отримання фізичної адреси за віртуальною адресою. З тією ж метою розмір сторінки вибирається рівним степені двійки, завдяки чому двійковий запис адреси легко розділяється на номер сторінки і зміщення, і в результаті в процедурі перетворення адрес триваліша операція складання замінюється операцією приєднання (конкатенації).

Розглянемо приклад, що пояснює основні характеристики організації сторінкової віртуальної пам'яті. Нехай комп'ютер має оперативну пам'ять об'ємом Еоп = 256 Мб, розмір сторінки вибраний рівним Естр = 4 Кб. У цьому випадку кількість фізичних сторінок рівна

Nf = Еоп / Естр = $256*20^{20} / 4*2^{10}$ = 64000 сторінок.

Для відображення фізичної адреси довільного байта оперативної пам'яті знадобиться $K = \log_2 256*20^{20} = 28$ двійкових розрядів.

Число розрядів для відображення зміщення в сторінці

$M = \log_2 4$ Кб $= \log_2 4096 = 12$.

Якщо процесор має 32-розрядну структуру, то на номер віртуальної сторінки відводиться 32-12=20 двійкових розрядів. Таким чином, число віртуальних сторінок дорівнює $N_в = 2^{20}$ (приблизно 1 млн віртуальних сторінок).

На адресу фізичної сторінки в нашому прикладі слід виділити 32-12 = 20 двійкових розрядів. На додаткові розряди, що характеризують властивості сторінки, знадобиться 1 байт. З іншого боку, немає необхідності в записі (дескрипторі) віртуальної сторінки мати поле з номером віртуальної сторінки (20 розрядів), оскільки адресу потрібного запису можна обчислювати, як це було розглянуто вище. Отже, в нашому прикладі довжина запису має бути рівною 32 - 12 + 8 = 28 двійковим розрядам, тобто з округленням до цілого числа байт – 4 байт. Таким чином, для кожного процесу, що виконується в комп'ютері, ОС створює таблицю сторінок розміром $4*N_в$ байт = $4*2^{20}$ = 4 Мб (рис. 10.10).

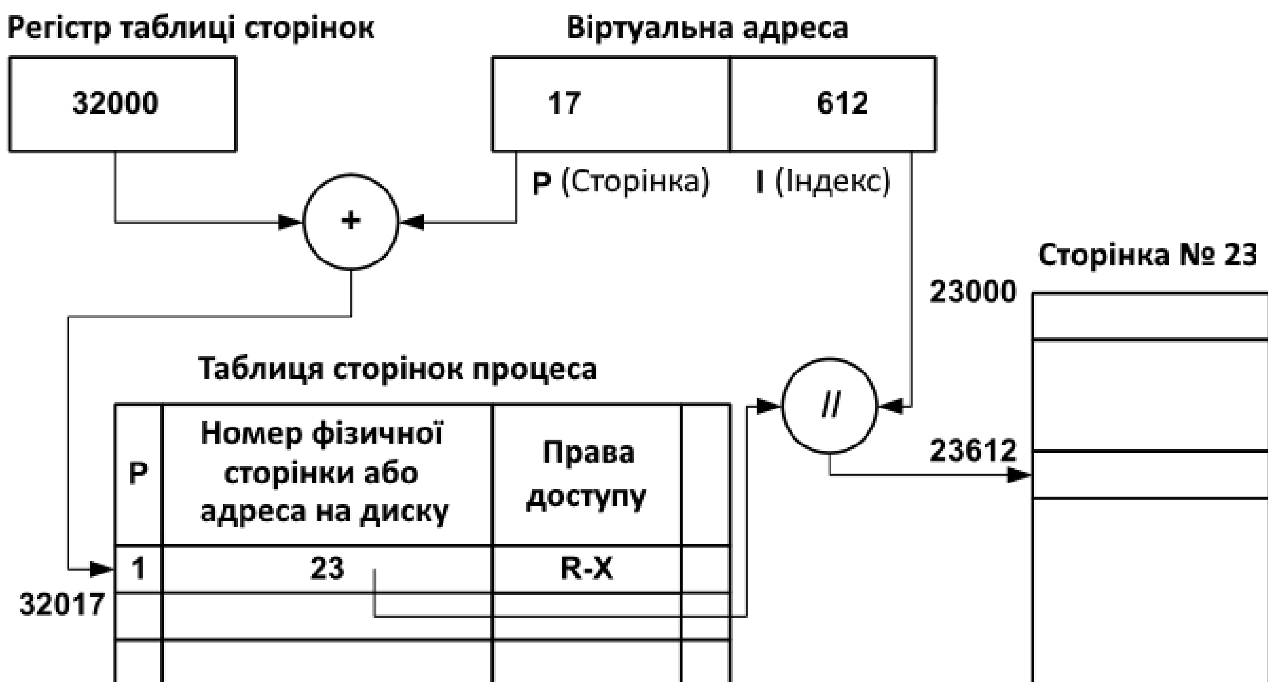


Рисунок 10.10 –Приклад розміщення сторінки в пам'яті

Іншим важливим чинником, що впливає на продуктивність системи, є частота сторінкових переривань, на яку, у свою чергу, впливають розмір сторінки і прийняті в цій системі правила вибору сторінок для вивантаження і завантаження. При неправильно вибраній стратегії заміщення сторінок можуть виникати ситуації, коли система витрачає велику частину часу даремно, на підкачування сторінок з оперативної пам'яті на диск і назад.

Щоб зменшити частоту сторінкових переривань, слід було б збільшувати розмір сторінки. Крім того, збільшення розміру сторінки зменшує розмір таблиці сторінок, тобто зменшує витрати пам'яті.

З іншого боку, якщо сторінка велика, то велика і фіктивна область в останній віртуальній сторінці кожної програми. В середньому на кожній програмі втрачається половина об'єму сторінки, що в сумі при великій сторінці може скласти істотну величину. З наведених міркувань виходить, що вибір розміру сторінки є складною оптимізаційною задачею, що вимагає обліку багатьох чинників.

На практиці ж розробники ОС і процесорів обмежуються деяким раціональним рішенням, придатним для широкого класу обчислювальних систем. Типовий розмір сторінки складає декілька кілобайт. Наприклад, найпоширеніші процесори x86 і Pentium компанії Intel, а також операційні системи, що встановлюються на цих процесорах, підтримують сторінки розміром 4096 байт (4 Кб). Так, процесор Pentium дозволяє використати також сторінки розміром до 4 Мб одночасно із сторінками об'ємом 4 Кб.

Час перетворення віртуальної адреси у фізичну значною мірою визначається часом доступу до таблиці сторінок. У зв'язку з цим таблицю сторінок прагнуть розміщати в "швидких" запам'ятовуючих пристроях. Це може бути, наприклад, набір спеціальних регістрів або пам'ять, що використовує для зменшення часу доступу: ***асоціативний пошук*** і ***кешування даних***.

Сторінковий розподіл пам'яті може бути реалізований в спрощеному варіанті, без вивантаження сторінок на диск. В цьому випадку усі віртуальні сторінки усіх процесів постійно знаходяться в оперативній пам'яті. Такий варіант сторінкової організації хоча і не надає користувачеві переваг роботи з віртуальною пам'яттю великого об'єму, але зберігає інші переваги сторінкової організації – дозволяє успішно боротися з фрагментацією фізичної пам'яті.

Дійсно, по-перше, програму можна розбити на частини і завантажити їх в розрізнені ділянки вільної пам'яті, а, по-друге, при завантаженні віртуальних сторінок ніколи не утворюється невикористаних залишків, оскільки розміри віртуальних і фізичних сторінок співпадають. Такий режим роботи системи управління пам'яттю використовується в деяких спеціалізованих ОС, коли потрібно забезпечити високу реактивність системи і здатність виконувати змінний набір додатків (приклад – ОС сімейства Novell NetWare 3.x і 4.x).

Нині відомі ще декілька методів підвищення ефективності функціонування сторінкової віртуальної пам'яті. До них належать:

– складніша структуризація віртуального адресного простору, наприклад, дворівнева (типова для 32-бітової адресації);

- використання спеціального високошвидкісного кеша для зберігання частини записів таблиці сторінок, який називають буфером швидкого перетворення адреси, або буфером пошуку трансляції (translation lookaside buffer – TLB);
- вибір оптимального розміру сторінки віртуальної пам'яті.

Зупинимося на можливостях реалізації цих методів.

10.8 Багаторівневі таблиці сторінок

При сторінковому розподілі пам'яті розмір сторінки впливає на кількість записів в таблицях сторінок. Чим менше сторінки, тим об'ємнішими є таблиці сторінок процесів і тим більше місця вони займають в пам'яті. Оскільки в сучасних 32-розрядних процесорах максимальний об'єм віртуального адресного простору процесу може досягати до 4 Гб (2^{32}), то при розмірі сторінки 4 Кб (2^{12}) і довжині запису 4 байти для зберігання таблиці сторінок може знадобитися 4 Мб пам'яті.

Виходом в такій ситуації є зберігання в пам'яті тільки тієї частини таблиці сторінок, яка активно використовується в цей період часу. Оскільки сама таблиця сторінок зберігається в таких же сторінках фізичної пам'яті, що і описувані нею сторінки, то принципово можливо тимчасово витісняти частину таблиці сторінок з оперативної пам'яті. Це означає, що самі таблиці сторінок стають об'єктами сторінкової організації, як і будь-які інші сторінки.

Щоб обійти проблему необхідності постійного зберігання в пам'яті величезних таблиць сторінок деякі процесори використовують дворівневу таблицю сторінок. При такій схемі є ***каталог таблиць сторінок (розділів)***, в якому кожен запис вказує на таблицю сторінок. Таким чином, якщо розмір каталогу – X, а максимальний розмір таблиці – Y, то процес може складатися максимум з X*Y сторінок. Максимальний розмір таблиці сторінок визначається умовою її розміщення в одній сторінці (як в Pentium).

На рис. 10.11 наведений приклад дворівневої схеми, типової для 32-бітової адресації. Подібна схема дозволяє істотно зберегти розмір призначеної для користувача таблиці сторінок, що розміщується в основній пам'яті (з 4 Мб до 4 Кб). Усі сторінки мають однаковий розмір, а розділи містять однакову кількість сторінок. Якщо розмір сторінки і кількість сторінок у розділі вибрати рівними мірі двійки (2^k і 2^n відповідно), то приналежність віртуальної адреси до розділу і сторінки, можна визначити дуже просто. Молодші **k** двійкових розрядів дають зміщення, наступні **n** розрядів є номером віртуальної сторінки, а старші розряди (позначимо їх кількість **m**), що залишилися, містять номер розділу.

Для кожного розділу будується власна таблиця сторінок. Кількість дескрипторів в таблиці і їх розмір підбираються такими, щоб об'єм таблиці виявився рівним об'єму сторінки. Наприклад, у процесорі Pentium при розмірі сторінки 4 Кб довжина дескриптора сторінки складає 4 байти, а кількість записів в таблиці сторінок, що поміщається на сторінку, дорівнює 1024.

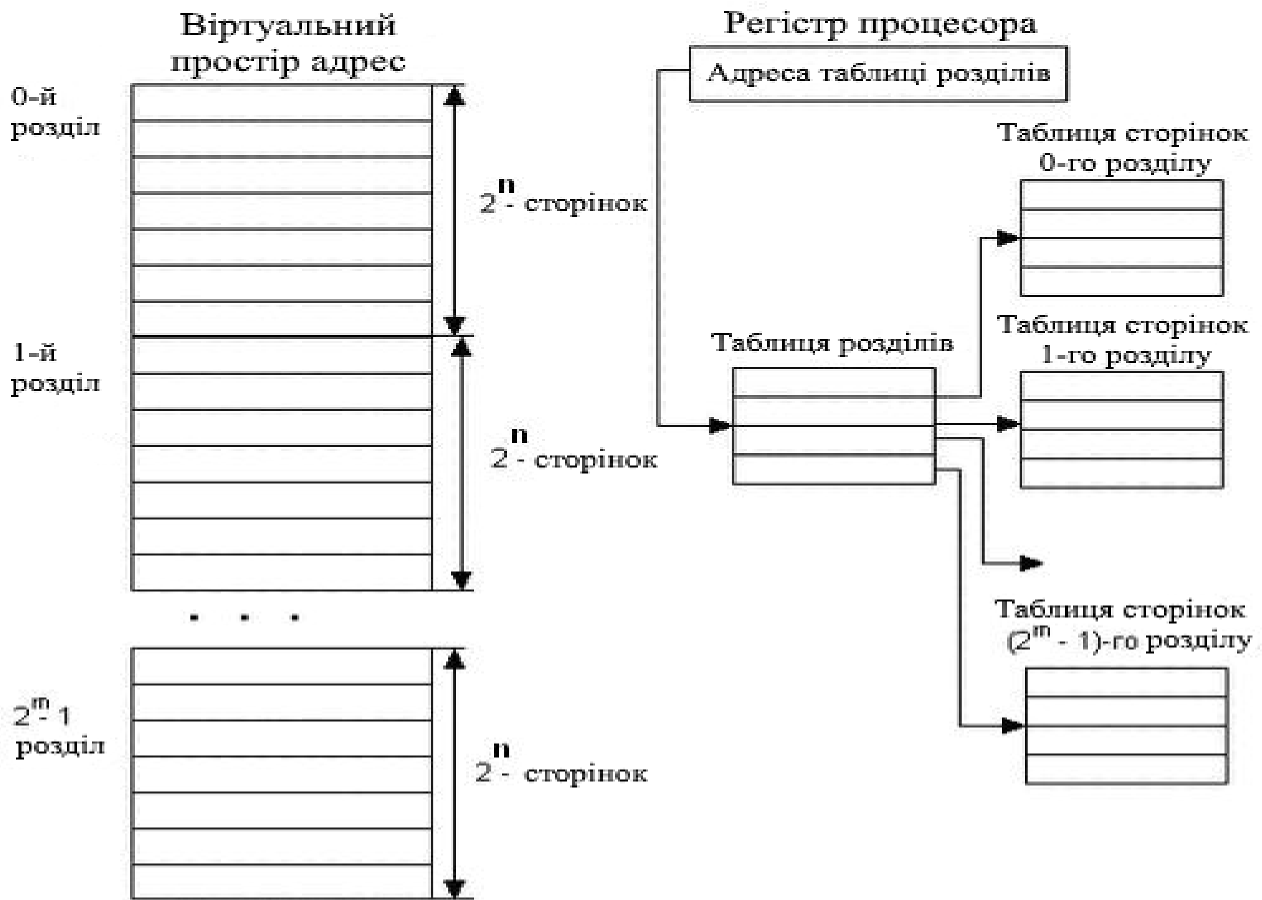


Рисунок 10.11 – Структура віртуального адресного простору з розділами

Ці дескриптори зведені в ***таблицю розділів*** (***каталог сторінок)***. Фізична адреса таблиці розділів активного процесу міститься в спеціальному регістрі процесора і тому завжди відома операційній системі. Сторінка, що містить таблицю розділів (коренева таблиця), ніколи не вивантажується з основної пам'яті, інакше робота віртуальної пам'яті була б неможлива.

Вивантаження сторінок з таблицями сторінок дозволяє заощадити пам'ять, але при цьому призводить до додаткових часових витрат при отриманні фізичної адреси. Дійсно, може статися так, що та таблиця сторінок, яка містить потрібний дескриптор, в даний момент вивантажена на диск, тоді процес перетворення адреси призупиняється до тих пір, поки необхідна сторінка не буде знову завантажена в пам'ять. Простежимо детальніше схему перетворення адрес для випадку дворівневої структуризації віртуального адресного простору (рис. 10.12):

1. Шляхом відкидання **k+n** молодших розрядів у віртуальній адресі визначається номер розділу, до якого належить ця віртуальна адреса.
2. По цьому номеру з таблиці розділів витягається дескриптор відповідної таблиці сторінок. Перевіряється, чи знаходиться ця таблиця сторінок в пам'яті. Якщо ні, відбувається сторінкове переривання і система завантажує потрібну сторінку з диска.
3. Далі з цієї таблиці сторінок витягається дескриптор віртуальної сторінки, номер якої міститься в середніх **n** розрядах перетворюваної віртуальної

адреси. Знову виконується перевірка наявності цієї сторінки в пам'яті і при необхідності її завантаження.

4. З дескриптора таблиці сторінок визначається номер (базова адреса) фізичної сторінки, в яку завантажена ця віртуальна сторінка. До номера фізичної сторінки пристиковується зміщення, взяте з **k** молодших розрядів віртуальної адреси. У результаті виходить шукана фізична адреса.

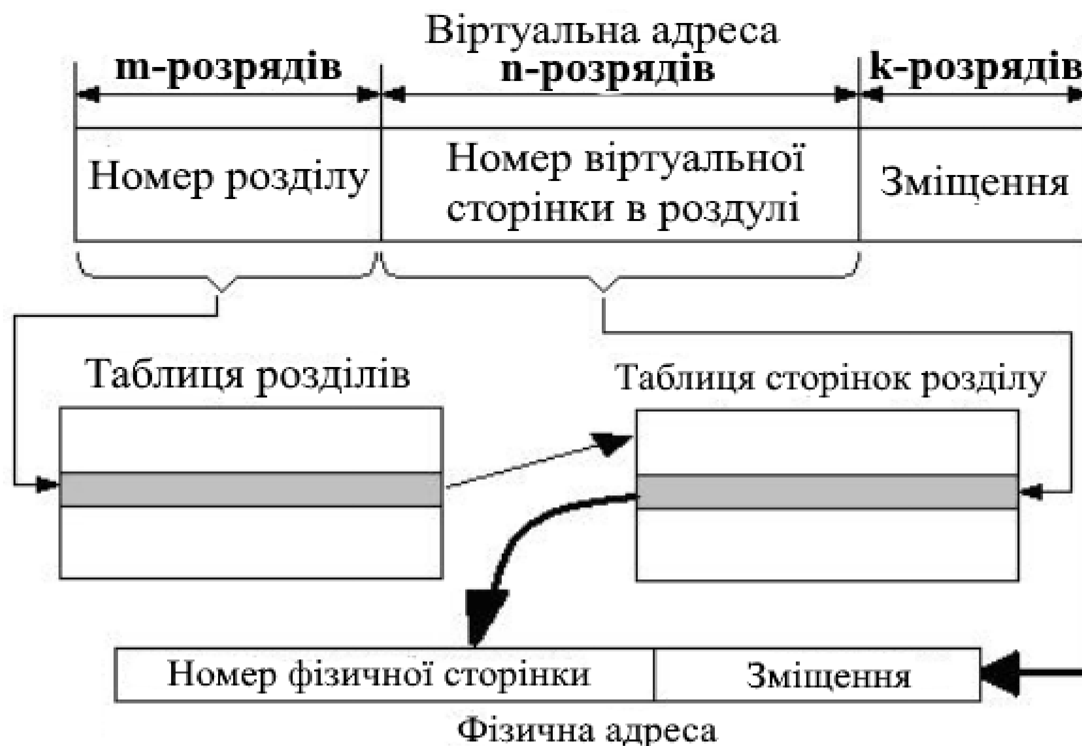


Рисунок 10.12 – Схема перетворення віртуальної адреси для дворівневої структуризації адресного простору

На рис. 10.13 наведений приклад дворівневої схеми, типової для 32-бітової адресації. Подібна схема дозволяє істотно зберегти розмір таблиці сторінок користувача, що розміщується в основній пам'яті. В даному випадку віртуальний адресний простір для процесу користувача може складати 2^{32} = 4 Гб. При об'ємі сторінки 2^{12} = 4 Кб в цьому просторі розміщується $2^{32}/2^{12} = 2^{20}$ сторінок. Таким чином, таблиця сторінок користувача матиме 2^{20} 4-байтних записів загальним об'ємом 4 Мб.

В основній пам'яті постійно знаходиться коренева таблиця, що містить 1024 записів, які вказують на початкову адресу призначеної для користувача таблиці сторінок (її об'єм, як вказано вище, 4 Мб). Вказівка на початкову адресу кореневої таблиці (активного процесу) заноситься в регістр процесора. Перші 10 біт віртуальної адреси використовуються для індексації в кореневій таблиці при пошуку записів про сторінку таблиці.

Якщо сторінка знаходиться в ОП, то наступні 12 біт віртуальної адреси використовуються для задання зміщення у фізичній сторінці ОП. Інакше генерується сторінкове переривання, але вже через відсутність потрібної сторінки процесу в ОП.

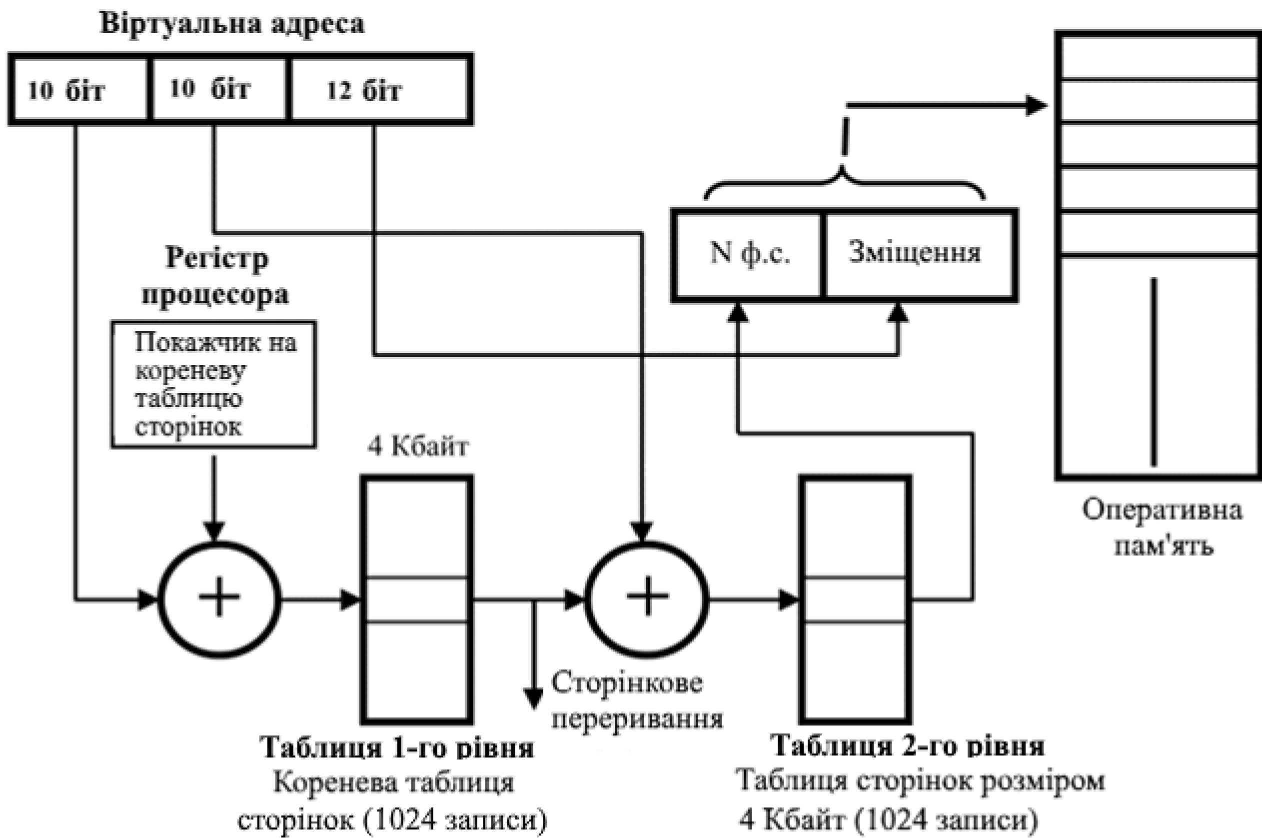


Рисунок 10.13 – Дворівнева схема таблиць сторінок

Таким чином, дворівнева схема, скорочуючи об'єм пам'яті для зберігання таблиці сторінок, в загальному випадку уповільнює перетворення віртуальної адреси із-за більшого числа можливих сторінкових переривань. Навіть якщо немає сторінкового переривання, потрібно три звернення до ОП замість двох при однорівневій сторінковій організації.

10.9 Буфери швидкого перетворення адреси (TLB)

Як уже відзначалося, проста схема сторінкової віртуальної пам'яті, по суті, подвоює час звернення до пам'яті: одне звернення для вибірки відповідного запису з таблиці сторінок, і ще одне звернення до адресних даних. А у разі використання дворівневих таблиць сторінок потрібні три операції доступу: до каталогу сторінок, до таблиці сторінок і безпосередньо за фізичною адресою. Для подолання цієї проблеми більшість схем віртуальної пам'яті використовують спеціальний високошвидкісний кеш для записів таблиці сторінок.

Практика використання віртуальної пам'яті показала, що для неї справедливий закон локалізації більшості звернень в невелику кількість нещодавно використаних сторінок (правило 90% до 10%), тому активно використовується тільки невелика частина таблиці сторінок.

Природне рішення проблеми прискорення – забезпечити комп'ютер апаратним пристроєм для відображення частини віртуальних сторінок у фізичні без звернення до таблиці сторінок. Тобто, мати невелику, швидку кеш-пам'ять, що зберігає необхідну на даний момент частину таблиці сторінок.

Модель локальності – принцип, покладений в роботу кеша. Якби доступ до будь-яких типів даних був випадковим, кеш був би даремний.

Для вирішення цієї проблеми більшість схем віртуальної пам'яті використовують спеціальний високошвидкісний кеш для записів таблиць сторінок, який називають ***буфером швидкого перетворення адреси,*** або ***буфером пошуку трансляції*** (translation lookaside buffer – **TLB**), або ***буфером асоціативної трансляції,*** або іноді ***асоціативною пам'яттю***.

Грунтуючись на правилі «***дев'яносто до десяти» (правилі локалізації***), що більшість програм схильна робити величезну кількість звернень до невеликої кількості сторінок, комп'ютер забезпечується невеликим апаратним пристроєм, що служить для відображення віртуальних адрес у фізичні без проходу по таблиці сторінок. Цей пристрій знаходиться усередині диспетчера пам'яті і складається з декількох елементів, від 8 до 4096. Так, в архітектурі Intel-32 таких елементів до Pentium-4 було 32 (що забезпечувало 98% попадань в кеш), починаючи з Pentium-4 – 128.

Оскільки ***асоціативна пам'ять*** містить тільки деякі із записів таблиці сторінок, то кожен запис в TLB повинен включати поле з номером віртуальної сторінки. Окрім цього, кожен запис таблиці асоціативної пам'яті містить інформацію про одну віртуальну сторінку, а саме: біт зміни сторінки, код захисту (читання/запис/виконання), біт дійсності запису (чи використовується вона в даний момент) і номер фізичного сторінкового блоку.

Пам'ять називається ***асоціативною***, оскільки в ній відбувається одночасне порівняння номера віртуальної сторінки, що відображається, з відповідним полем в усіх рядках цієї невеликої таблиці.

Коли віртуальна адреса представляється диспетчером пам'яті для відображення, апаратура спочатку переконується в тому, що номер його віртуальної сторінки є присутнім у буфері TLB шляхом порівняння адреси з усіма записами одночасно. Якщо знайдено співпадіння, то сторінковий блок береться прямо з буфера, без переходу до таблиці сторінок (рис. 10.14).

Якщо номер віртуальної сторінки не знаходиться в TLB, то диспетчер пам'яті виконує звичайний пошук в таблиці сторінок. Потім він видаляє один із записів з буфера (зазвичай старіший елемент) і замінює його тільки що знайденим записом з таблиці сторінок, з ймовірністю того, що цей запис знову незабаром зажадається. Тут ми стикаємося з традиційною для будь-якого кеша проблемою заміщення, а саме який із записів в кеші необхідно змінити. Конструкція асоціативної пам'яті повинна організовувати записи так, щоб можна було прийняти рішення про те, який із старих записів має бути видалений при внесенні нових.

Основні принципи, за якими організується робота асоціативної пам'яті, показані на рис. 10.15. Процесор апаратно здатний одночасно опитувати всі записи TLB для визначення того, яка з них відповідає заданому номеру сторінки. Такий підхід відомий як асоціативне відображення (associative mapping), на відміну від прямого відображення, або індексування, що застосовується для пошуку в таблиці сторінок.

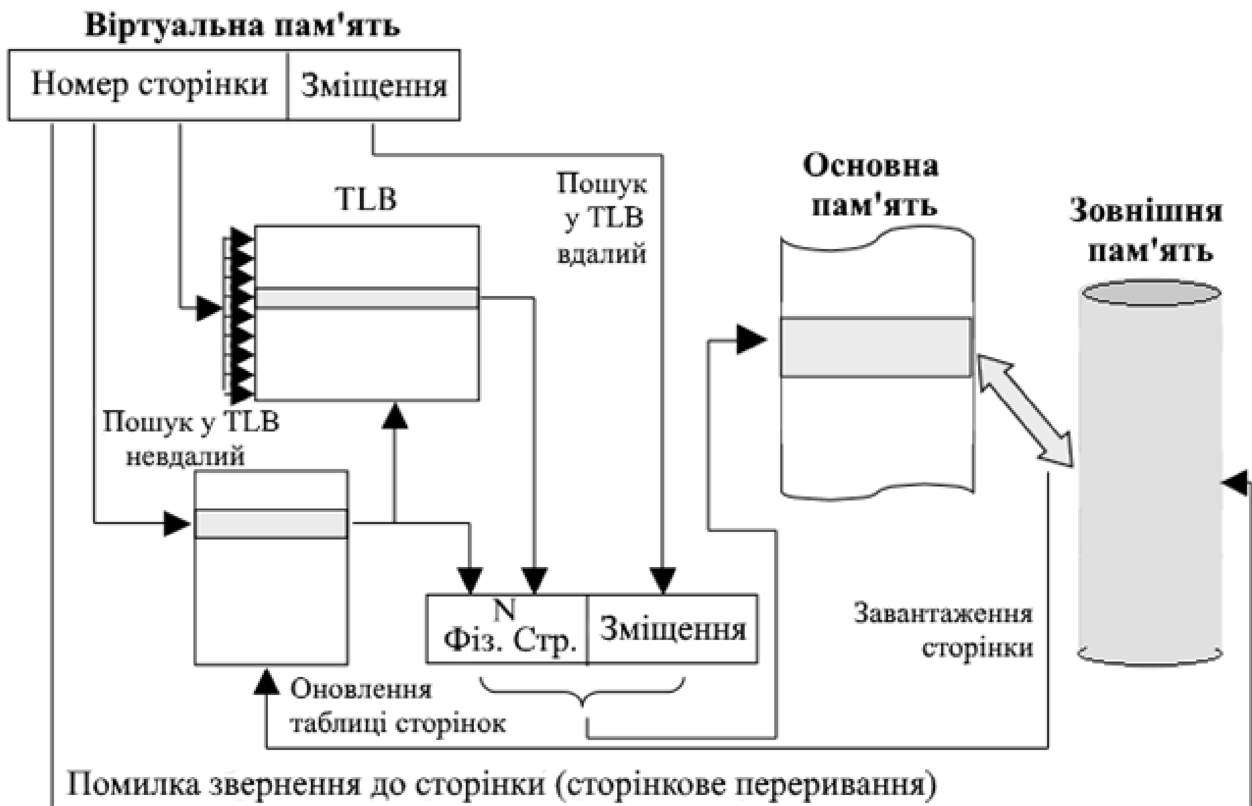


Рисунок 10.14 – Буфер швидкої переадресації

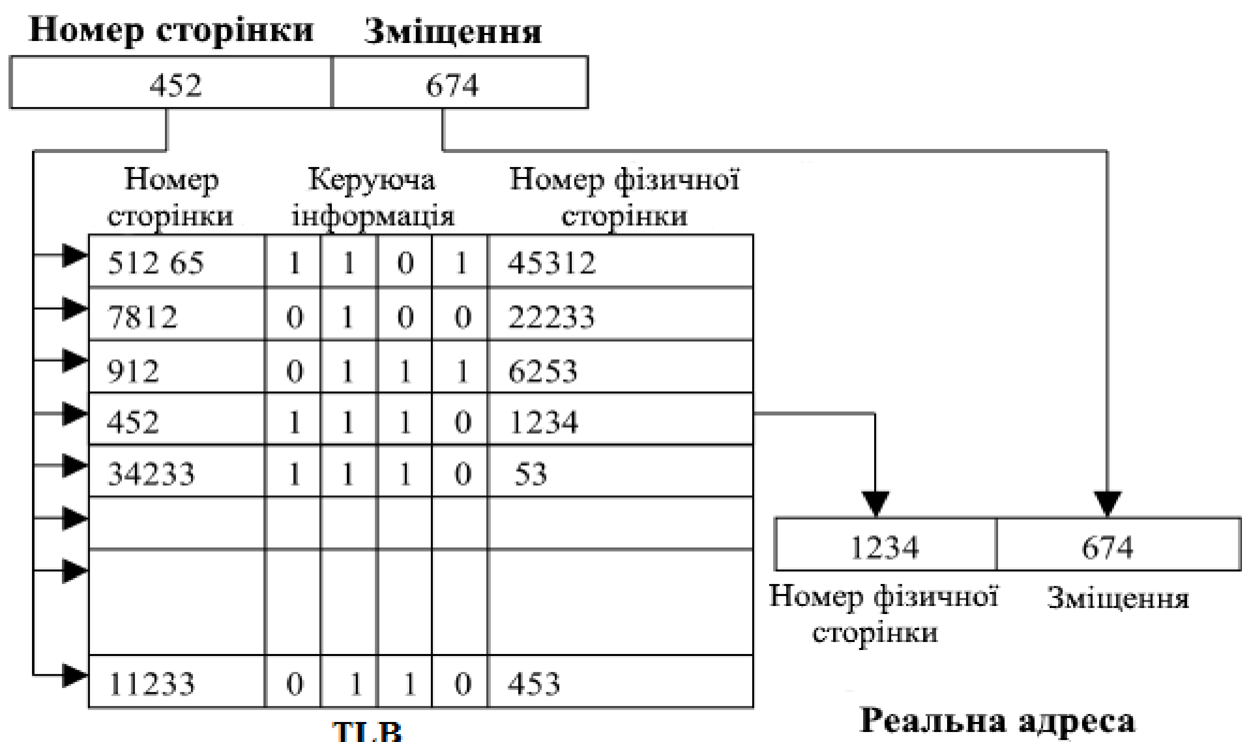


Рисунок 10.15. Асоціативна пам'ять

Число вдалих пошуків номера сторінки в ***асоціативній пам'яті*** по відношенню до загального числа пошуків називається ***hit*** (співпадіння) ***ratio*** (пропорція, відношення). Іноді також використовується термін **«*відсоток***

попадань в кеш». Звернення до одних і тих же сторінок підвищує *hit ratio*. Чим більше *hit ratio*, тим менше середній час доступу до даних, що знаходяться в оперативній пам'яті.

Припустимо, наприклад, що для визначення адреси в разі кеш-промаху через таблицю сторінок потрібно **100 нс (t1)**, а для визначення адреси в разі кеш-попадання через асоціативну пам'ять – **20 нс(t2).** З 90% *hit ratio* (***p*** – ймовірність кеш-попадания) середній час визначення адреси за формулою повної ймовірності: **t = t1 + t2*p = 100*0,1 + 20*0,9 = 28 нс.**

У разі перемикання контексту в архітектурі Intel-32 необхідно очистити увесь кеш, оскільки для кожного процесу є своя таблиця сторінок, і ті ж самі номери сторінок для різних процесів можуть відповідати різним фреймам у фізичній пам'яті. Таким чином, використання асоціативної пам'яті збільшує час перемикання контексту. Дворівнева (асоціативна пам'ять + таблиця сторінок) схема перетворення адреси є яскравим прикладом ієрархії пам'яті, заснованої на використанні принципу локальності, про що ми говоритимемо в наступному розділі.

Слід підкреслити, що механізм віртуальної пам'яті повинен взаємодіяти з **кешем оперативної пам'яті (**окрім TLB). Ця взаємодія показана на рис. 10.16 [10].

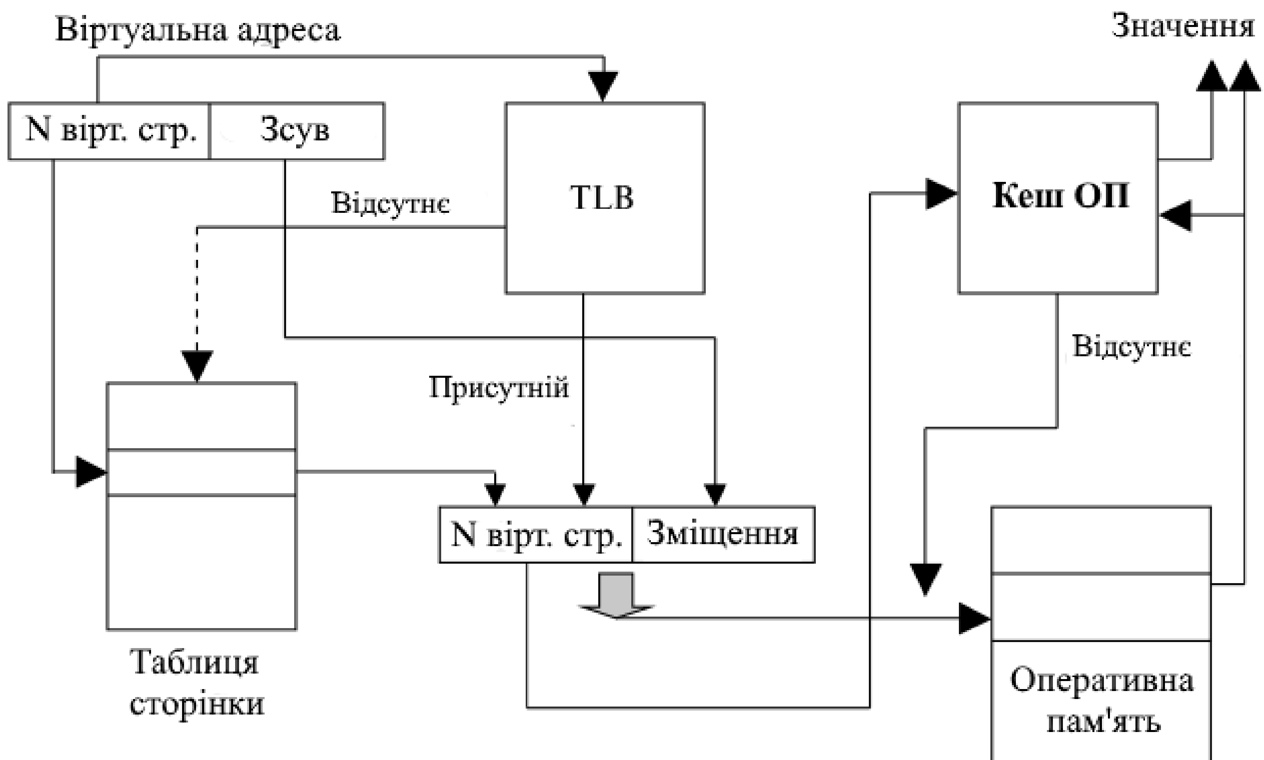


Рисунок 10.16. Використання кешу оперативної пам'яті

Спочатку відбувається звернення до TLB для з'ясування наявності в ньому відповідного запису таблиці сторінок. При позитивному результаті шляхом об'єднання номера фізичної сторінки, що отримується з TLB, і зміщення генерується фізична адреса. Якщо необхідного запису в TLB немає, вона вибирається з таблиці сторінок. Після отримання фізичної адреси в обох

ситуаціях виконується звернення до кеша оперативної пам'яті для з'ясування наявності в ньому блоку з необхідною фізичною адресою. Якщо відповідь позитивна, то необхідне значення (код або дані) передається процесору. Інакше робиться вибірка слова з основної пам'яті і оновлюється вміст кеша основної пам'яті.

10.10 Інвертовані таблиці сторінок, хеш-таблиці

Традиційні таблиці сторінок вимагають по одному запису на кожну сторінку. Якщо адресний простір складається з 2^{32} байт з розміром сторінки 4096 байт(2^{12}), тоді таблиця сторінок містить більше мільйона записів і займатиме мінімум 4 Мб (2^{20}). Зрозуміло, що виділяти таку кількість оперативної пам'яті під таблиці сторінок недоцільно. А при 64-розрядному адресному просторі з розміром сторінки 4 Мб і розміром запису таблиці в 8 байт, таблиця займе більше 30 Тб (біля 4 триліонів записів). Це нереально навіть у майбутньому.

Тому ще одним підходом до використання одно- або дворівневих таблиць сторінок являється застосування ***інвертованої таблиці сторінок*** (рис. 10.17). Цей підхід застосовується на машинах PowerPC, деяких робочих станціях Hewlett-Packard, IBM RT, IBM AS/400 і ряді інших. Особливо інвертовані таблиці сторінок знайшли широке застосування на 64-розрядних машинах.

У цій моделі таблиця містить по одному запису на кожен сторінковий кадр фізичної пам'яті, а не на сторінку у віртуальному адресному просторі. Істотно, що достатньо однієї таблиці для усіх процесів. Наприклад, при 64-розрядних віртуальних адресах, при розмірі сторінок 4 Кб і 1 Гб оперативної пам'яті інвертована таблиця сторінок зажадає усього лише 262144 записів. Кожен запис інвертованої таблиці сторінок містить номер процесу і номер віртуальної сторінки. Слід зауважити, що інвертовані сторінкові таблиці не зберігають інформації про розміщення нерезидентних сторінок на вторинних пристроях зберігання. Ця інформація підтримується ОС.

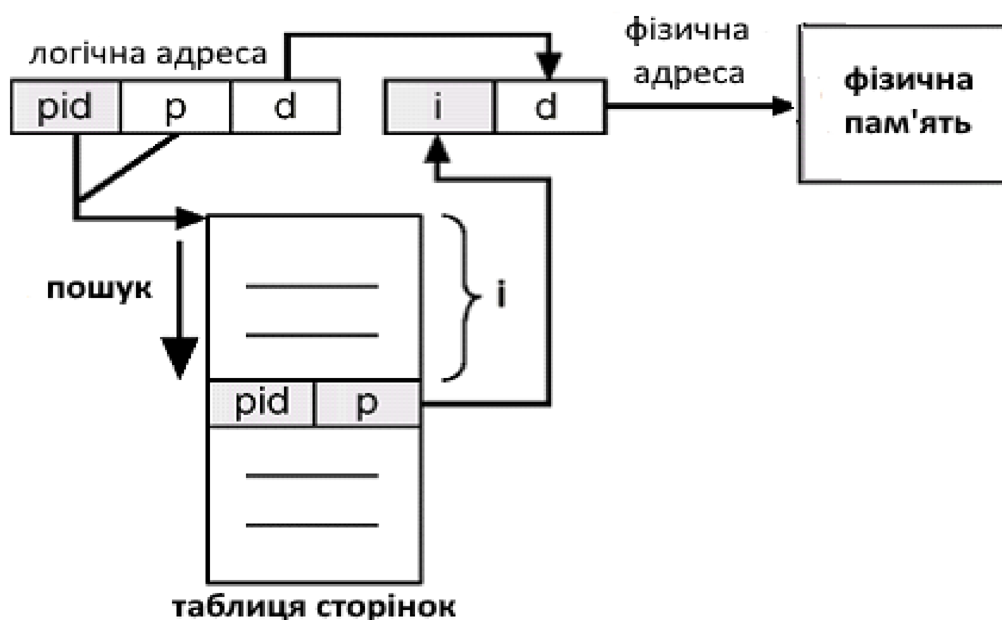


Рисунок 10.17 – Інвертована таблиця сторінок

Хоча інвертовані таблиці сторінок економлять значну кількість місця, вони мають серйозний недолік – записи в них (як і в *асоціативній пам'яті)* не відсортовані за збільшенням номерів віртуальних сторінок, що ускладнює трансляцію адреси. Коли процес *N* звертається до віртуальної сторінки *P,* апаратне забезпечення не може більше знайти фізичну сторінку, використовуючи номер *P* як індекс в таблиці сторінок. Замість цього здійснюється пошук запису (***pid, P)*** в усій інвертованій таблиці сторінок, де ***pid*** – ідентифікатор процесу *N.* Вийти з цього положення можна, використовуючи розглянутий раніше буфер швидкого перетворення адреси (TLB). Але при невдалому пошуку в буфері TLB пошук в інвертованій таблиці сторінок повинен виконуватися програмно. Один з можливих способів удосконалити його – підтримувати ***хеш-таблицю*** віртуальних адрес.

При такому підході частина віртуальної адреси, яка представляє номер сторінки, відображається в хеш-таблицю з використанням простої функції хешування. Наприклад, для *N* елементів, які зберігаються в таблиці розміром $M \geq N$ (причому *M* не набагато більше *N*), мітка елементу перетвориться на майже випадкове число *n* між *0* і *M-1* методом ділення мітки по модулю *M.*

Цілочисельна функція ***hash*** визначає значення ***s*** з відрізку *[0, M - 1].* При пошуку елемента ***s*** спочатку обчислюється ***hash(s)***, а потім виконується пошук тільки в списку ***Shash(s).*** Оскільки на один і той же запис хеш-таблиці можуть відображатися декілька віртуальних адрес, то для обробки переповнення використовується технологія ланцюжків, які на практиці досить короткі – як правило, від одного до двох записів (рис. 10.18).

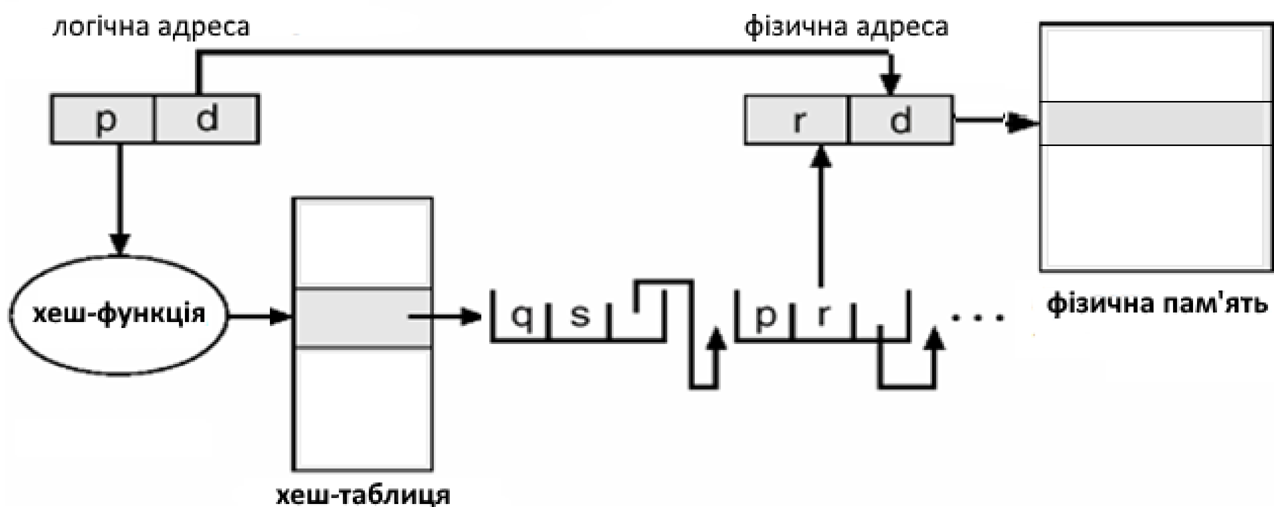


Рисунок 10.18 – Хеш-таблиця сторінок

Хеш-таблиця містить покажчик на інвертовану таблицю сторінок. Кожній сторінці реальної пам'яті при цьому відповідає один запис в хеш-таблиці і в інвертованій таблиці сторінок. Таким чином, для зберігання таблиць потрібна фіксована частина основної пам'яті, незалежно від розміру і кількості процесів і підтримуваних віртуальних сторінок.

Наприклад, вставка елементу в таблицю відбувається таким чином.

1. Перетворимо мітку елементу в майже випадкове число *s між 0 і M-1* методом ділення мітки по модулю *M.*
2. Використовуємо отримане значення *s* як індекс в хеш-таблиці:

   А. Якщо відповідний запис в таблиці порожній, значить, елемент раніше не був збережений в таблиці.

   Б. Якщо запис вже зайнятий і її мітка відповідає шуканій, значить, знайдений необхідний елемент.

   В. Якщо запис зайнятий, і її мітка не відповідає заданій, продовжуємо пошук в області переповнювання.

Середня тривалість пошуку елемента при відкритій хеш-таблиці з використанням таблиць переповнювання з ланцюжками, рівна *1/2M,* для великих значень $N=M$ прагне до *1.5,* зокрема для бінарного пошуку в сортованому списку тривалість пошуку елементу рівна Log_2M.

10.11 Визначення найкращого розміру сторінки

При виборі розміру сторінки треба враховувати декілька чинників. Один з них – внутрішня фрагментація, яка безпосередньо залежить від розміру сторінки. Внутрішня фрагментація зменшується зі зменшенням розміру сторінки. Проте чим менше сторінка, тим більше їх потрібно для процесу, що означає збільшення розміру таблиці сторінок. Розмір сторінки впливає і на частоту виникнення переривання через відсутність сторінки в основній пам'яті.

Реально розміри сторінок різних комп'ютерів складають такі значення: 512 байт (сімейство VAX, IBM AS/400), 4 Кб (IBM 370), 8 Кб (DEC Alpha), від 4 Кб до 4 Мб (Pentium).

Визначення найкращого розміру сторінки вимагає урівноваження декількох чинників. Тому не існує абсолютного оптимального рішення. Існує два аргументи на користь маленького розміру сторінок. Випадково вибраний текст, дані або сегмент стека не заповнюють цілу кількість сторінок. В середньому половина останньої сторінки виявляється порожньою. Якщо в пам'яті *n* сегментів при розмірі сторінки *p* байт, то *np/2* байт буде витрачений даремно в результаті внутрішньої фрагментації. Це розумний аргумент на користь сторінок невеликого розміру.

Інший аргумент стає очевидним, якщо представити програму, що складається з восьми послідовних етапів, по 4 Кб кожен. При розмірі сторінки 32 Кб програмі мають бути постійно виділені 32 Кб. При розмірі сторінки 16 Кб їй потрібні тільки 16 Кб. При розмірі сторінки 4 Кб, або менше, програма вимагає усього лише 4 Кб у будь-який момент часу. Тобто, великий розмір сторінки швидше, чим маленький, стане причиною того, що в пам'яті знаходиться невживана частина сторінки.

З іншого боку, невеликий розмір сторінки означає, що програмам буде потрібно багато сторінок, отже потрібна величезна таблиця сторінок. Як правило, сторінка за раз переноситься на диск і з нього. При цьому велика частина часу йде на пошук циліндра і затримку обертання. Так що переміщення маленької сторінки займає майже стільки ж часу, скільки і великої. Може

знадобитися 64*10 мс, щоб завантажити 64 сторінки розміром 512 байт, і усього лише 4*12 мс для завантаження 4-х сторінок по 8 Кб.

Враховуючи усе це можна математично проаналізувати розмір сторінки. Нехай середній розмір процесу рівний ***S*** байт, а сторінки – ***P*** байт. Крім того, припустимо, що запис для кожної сторінки вимагає ***E*** байт. Тоді приблизна кількість сторінок, необхідна для процесу, рівна ***S/P***, що займе ***SE/P*** байт для таблиці сторінок. Втрата пам'яті в останній сторінці процесу рівна ***P/2.*** Таким чином, загальні накладні витрати внаслідок підтримки таблиці сторінок і втрати від внутрішньої фрагментації дорівнюють сумі цих складових:

$$витрата = SE/P + P/2.$$

Перший доданок (розмір таблиці сторінок) збільшується при зменшенні розміру сторінок. Другий доданок (внутрішня фрагментація) при збільшенні розміру сторінок зростає. Оптимальний варіант повинен знаходитися десь посередині. Якщо узяти 1-у похідну по змінній ***P*** і прирівняти її до нуля, отримаємо рівність: $- SE/P2 + ½ = 0$. З цієї рівності можна отримати формулу, що дає оптимальний розмір сторінок (беручи до уваги тільки втрати пам'яті на фрагментацію і розмір таблиці сторінок). У результаті вийде: $P = \sqrt{2SE}$.

Для середнього розміру процесу ***S*** = 1 Мб і розміру запису в таблиці сторінок ***E*** = 8 байт оптимальний розмір сторінки дорівнюватиме 4 Кб. У сучасних комп'ютерах частіше зустрічаються розміри сторінок 4 Кб або 8 Кб. Оскільки пам'яті стає більше, то розмір сторінок також має тенденцію зростання (але залежність нелінійна).

10.12 Сегментна організація віртуальної пам'яті

При сторінковій організації віртуальний адресний простір процесу ділиться механічно на рівні частини. Це не дозволяє диференціювати способи доступу до різних частин програми (***сегментів***), а ця властивість часто буває дуже корисною. Наприклад, можна заборонити поводитися з операціями запису і читання в кодовий сегмент програми, а для сегменту даних дозволити тільки читання. Крім того, розбиття програми на «осмислені» частини робить принципово можливим розділення одного сегменту декількома процесами. Наприклад, якщо два процеси використовують одну і ту ж математичну підпрограму, то в оперативну пам'ять може бути завантажена тільки одна копія цієї підпрограми. Покращується захист сегментів, оскільки програміст може визначати певні ділянки програми або даних і призначати їм права доступу.

До сих пір розглянута віртуальна пам'ять була одновимірною, оскільки в ній адреси слідували одна за одною від 0 до деякого максимального значення. Але для вирішення багатьох проблем наявність двох і більше окремих віртуальних адресних просторів може бути раціональнішим варіантом, ніж наявність тільки одного адресного простору. Наприклад, у компілятора є декілька таблиць, що створюються в процесі компіляції, в які можуть входити:

1. Початковий текст, збережений для друку лістингу (у пакетних системах).
2. Таблиця імен, що містить імена і атрибути змінних.

3. Таблиця, що містить усі використовувані константи, як цілочисельні, так і з плаваючою точкою.
4. Дерево розбору, в якому міститься синтаксичний аналіз програми.
5. Стек, який використовується для викликів процедур усередині компілятора.

У процесі компіляції кожна з перших чотирьох таблиць постійно росте. А остання збільшується і зменшується в розмірах абсолютно непередбачуваним чином. В одновимірній пам'яті цим п'яти таблицям мають бути виділені послідовні ділянки віртуального адресного простору, показані на рис. 10.19 [9].

Рисунок 10.19 – Розміщення таблиць компілятора в одновимірному адресному просторі

Розглянемо, що вийде, якщо програма містить набагато більшу, ніж зазвичай, кількість змінних, але цілком звичайну кількість усіх інших компонентів. Ділянка адресного простору, виділена під таблицю імен, може заповнитися повністю, але для інших таблиць може залишитися велика кількість вільного простору. Зрозуміло, що компілятор може просто видати повідомлення про те, що він не може продовжити роботу через занадто велику кількість змінних, але усе це виглядатиме не занадто переконливо на тлі того, що у інших таблиць залишився невикористаний простір.

Існує також інша можливість: забрати простір у таблиць, що мають його в надлишку, і передати його таблицям з дефіцитом простору. У кращому випадку це принесе одні неприємності, а в гіршому – стомливу і нерозумну роботу. Насправді тут потрібний спосіб позбавити програміста від необхідності збільшення і зменшення розмірів таблиць, аналогічно тому, як віртуальна пам'ять усуває недоліки з приводу організації програми в оверлеї.

Простим і дуже універсальним рішенням є надання машини з великою кількістю абсолютно незалежних адресних просторів, що називаються

сегментами. Кожен сегмент складається з лінійної послідовності адрес від 0 до деякого максимуму. Довжина кожного сегменту може мати будь-яке значення від 0 до максимально дозволеного. Різні сегменти можуть бути різної довжини, як це звичайно і трапляється. Крім того, довжина сегменту може змінюватися в процесі виконання програми. Довжина сегменту стека може збільшуватися при записі в нього даних і зменшуватися при їх витяганні з нього.

Оскільки кожен сегмент містить окремий адресний простір, різні сегменти можуть розростатися або звужуватися незалежно, не впливаючи один на одного. Для вказівки адреси в такій сегментованій, або двовимірній пам'яті, програма повинна надати адресу, що складається з двох частин: номери сегменту і адреси усередині цього сегменту. На рис. 10.20 показано п'ять незалежних сегментів пам'яті, яка використовується для розглянутих раніше таблиць компілятора.

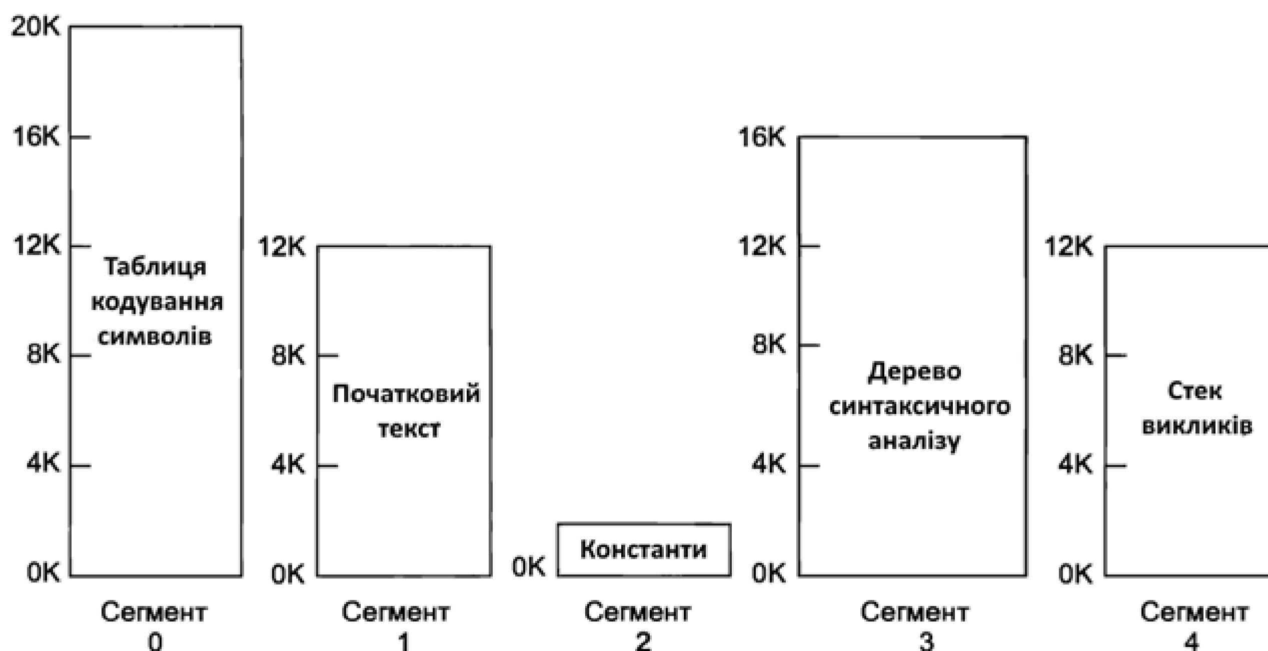


Рисунок 10.20 – Сегментована пам'ять з п'ятьма незалежними сегментами

Після того як усі процедури програми скомпільовані і скомпоновані, то у виклику процедури, яка звертається до процедури в сегменті *n*, буде використана адреса, що складається з двох частин (*n*, 0) і адресована до слова 0 (до точки входу).

Якщо згодом процедура в сегменті *n* буде змінена і перекомпільована, то змінювати інші процедури вже не доведеться (оскільки початкові адреси не будуть змінені), навіть якщо нова версія буде більше за стару. При використанні одновимірної пам'яті процедури компонуються безпосередньо одна за одною, без якого-небудь адресного простору між ними.

Отже, зміна розмірів однієї процедури вплине на початкові адреси інших (не пов'язаних з нею) процедур. А це, у свою чергу, зажадає зміни усіх процедур, з яких викликаються будь-які з переміщених процедур, щоб врахувати їх нові початкові адреси.

Сегментація також полегшує спільне використання процедур або даних декількома процесами. Типовим прикладом може послужити спільно

використовувана бібліотека. У сегментованій системі такі бібліотеки можуть бути поміщені в сегмент і спільно використовуватися декількома процесами, виключаючи потребу у своїй присутності в адресному просторі кожного процесу.

Оскільки кожен сегмент формує логічний об'єкт, наприклад процедуру, або масив, або стек, у різних сегментів можуть бути різні види захисту. Сегмент процедури може бути визначений тільки як виконуваний, із забороною спроб що-небудь в ньому прочитати або зберегти. Масив чисел з плаваючою точкою може бути визначений для читання і запису, але не для виконання, і спроби передати йому управління. Подібний захист дуже корисний при виявленні помилок програмування.

Розглянемо, яким чином сегментний розподіл пам'яті реалізує ці можливості (рис. 10.21). Віртуальний адресний простір процесу ділиться на сегменти, розмір яких визначається програмістом з урахуванням смислового значення інформації, що міститься в них. Окремий сегмент може бути підпрограмою, масивом даних і тому подібне. Іноді сегментація програми виконується за умовчанням компілятором.

При завантаженні процесу частина сегментів поміщається в оперативну пам'ять, а частина сегментів розміщується в дисковій пам'яті. Сегменти однієї програми можуть займати в оперативній пам'яті несуміжні ділянки.

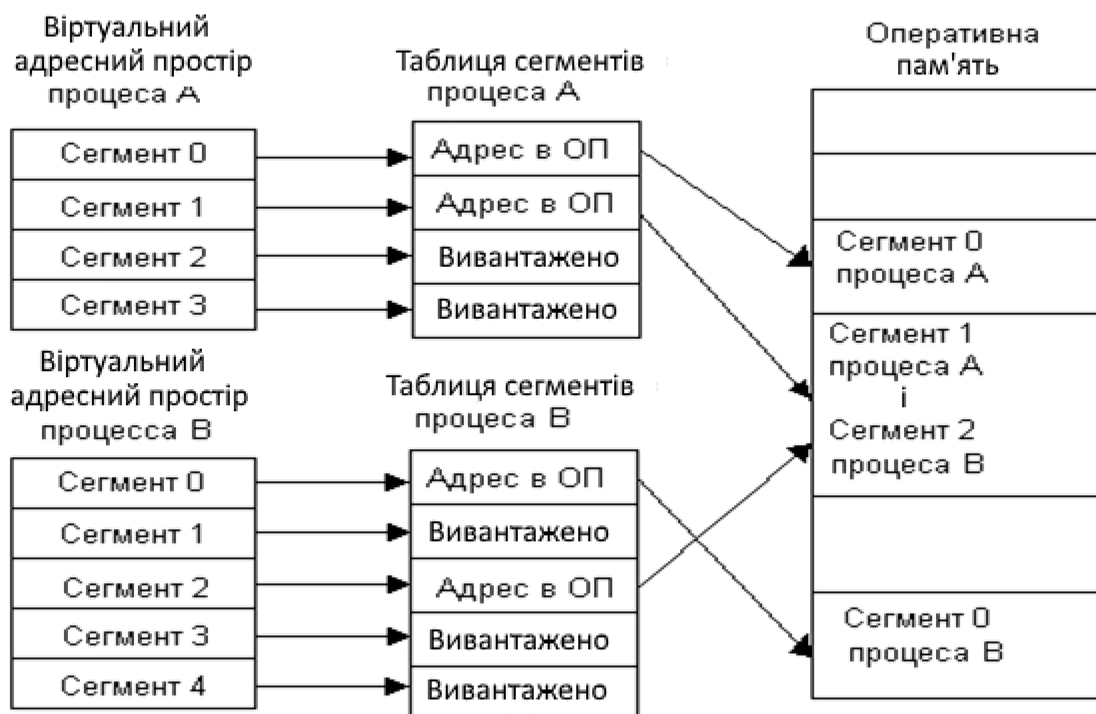


Рисунок 10.21 – Розподіл пам'яті сегментами

Під час завантаження система створює таблицю сегментів процесу (аналогічно таблиці сторінок), в якій для кожного сегменту вказується початкова фізична адреса сегменту в оперативній пам'яті, розмір сегменту, правила доступу, ознака модифікації, ознака звернення до цього сегменту за останній інтервал часу і деяка інша інформація (дескриптор сегмента).

Біт модифікації вказує, чи було змінено вміст сегменту з часу його останнього завантаження в основну пам'ять. Якщо змін не було, то при вивантаженні сегменту немає необхідності в його повторному записі на диск.

Якщо віртуальні адресні простори декількох процесів включають один і той же сегмент, то в таблицях сегментів цих процесів робляться посилання на одну і ту ж ділянку оперативної пам'яті, в яку цей сегмент завантажується в єдиному екземплярі. Система з сегментною організацією функціонує аналогічно системі із сторінковою організацією. Час від часу відбуваються переривання, пов'язані з відсутністю потрібних сегментів в пам'яті. При необхідності звільнення пам'яті деякі сегменти вивантажуються. При кожному зверненні до оперативної пам'яті виконується перетворення віртуальної адреси у фізичну адресу. Крім того, при зверненні до пам'яті перевіряється чи дозволений доступ необхідного типу до цього сегменту.

Віртуальна адреса при сегментній організації пам'яті може бути представлена парою (*g, s),* де *g* – номер сегменту, а *s* – зміщення в сегменті. Фізична адреса виходить шляхом складання початкової фізичної адреси сегменту, знайденого в таблиці сегментів за номером *g,* і зміщення *s* (рис. 10.22). У даному випадку не можна обійтися операцією конкатенації, як це робиться при сторінковій організації пам'яті.

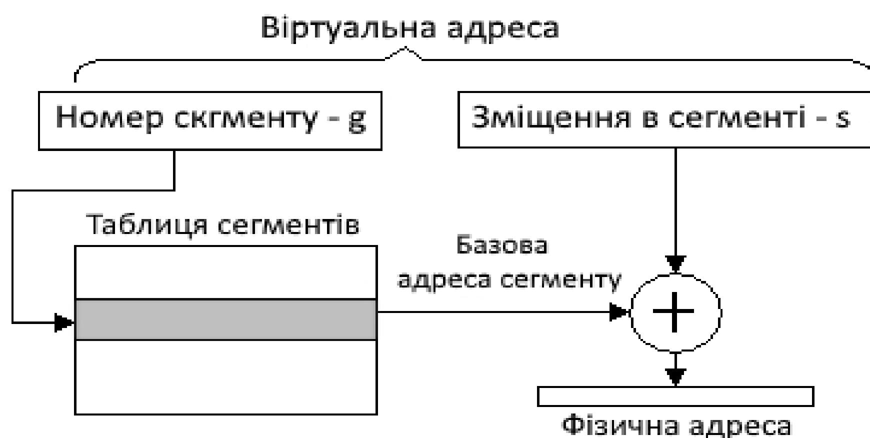


Рисунок 10.22 – Перетворення віртуальної адреси при сегментній організації пам'яті

Дійсно, оскільки розмір сторінки дорівнює степені двійки, то в двійковому виді він виражається числом з декількома нулями в молодших розрядах. Сторінки мають однаковий розмір, тобто їх початкові адреси кратні розміру сторінок і виражаються також числами з нулями в молодших розрядах. Саме тому ОС заносить в таблиці сторінок не повні адреси, а номери фізичних сторінок, які співпадають із старшими розрядами базових адрес. Сегмент же може в загальному випадку розташовуватися у фізичній пам'яті, починаючи з будь-якої адреси (байта). Отже, для визначення місця розташування в пам'яті необхідно задавати його повну ***початкову фізичну адресу.***

На рис. 10.23 зображений приклад звернення до елементу пам'яті, віртуальна адреса якого дорівнює сегменту з номером 11 і зміщенням від початку

цього сегменту, рівним 612. Операційна система розмістила цей сегмент у пам'яті, починаючи з комірки з номером 19700.

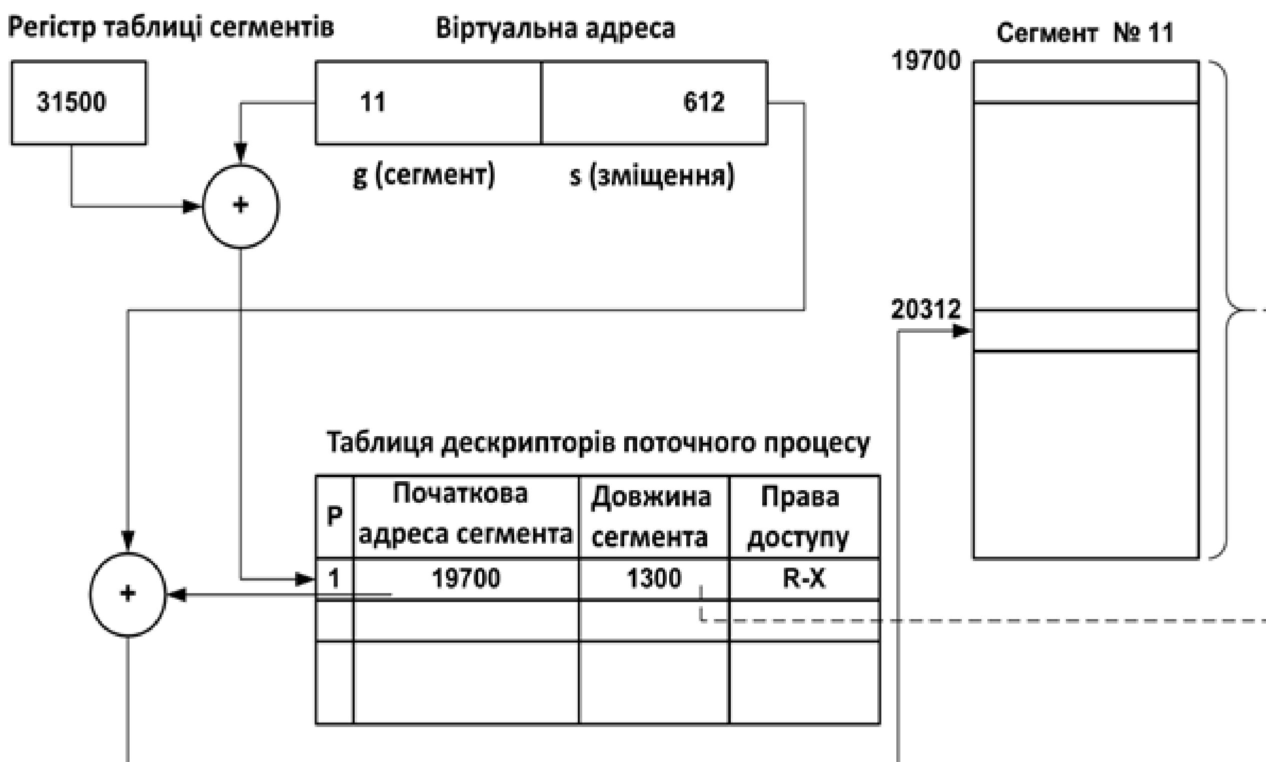


Рисунок 10.23 – Приклад розміщення сегменту в пам'яті

Використання операції складання замість конкатенації уповільнює процедуру перетворення віртуальної адреси у фізичну в порівнянні із сторінковою організацією.

Іншим недоліком сегментного розподілу є надмірність. При сегментній організації одиницею переміщення між пам'яттю і диском є сегмент, що має в загальному випадку об'єм більший, ніж сторінка. Проте в багатьох випадках для роботи програми зовсім не потрібно завантажувати увесь сегмент цілком, досить було б однієї або двох сторінок. Аналогічно за відсутності вільного місця в пам'яті не варто вивантажувати цілий сегмент, коли можна обійтися вивантаженням декількох сторінок.

Але головний недолік сегментного розподілу – це фрагментація на рівні сегментів (зовнішня фрагментація), яка виникає із-за непередбачуваності розмірів сегментів. У процесі роботи системи в пам'яті утворюються невеликі ділянки вільної пам'яті, в які не може бути завантажений жоден сегмент. Сумарний об'єм, зайнятий фрагментами, може скласти істотну частину загальної пам'яті системи, призводячи до її неефективного використання.

На рис. 10.24, *а* показаний приклад фізичної пам'яті, що спочатку має п'ять сегментів. Тепер розглянемо, що вийде, якщо сегмент 1 віддаляється, а на його місце поміщається менший за розміром сегмент 7. У нас вийде конфігурація пам'яті, показана на рис. 10.24, *б.* Між сегментом 7 і сегментом 2 буде невживана область, тобто діра. Потім сегмент 4 замінюється сегментом 5, як показано на рис. 10.24, *в,* а сегмент 3 замінюється сегментом 6, як показано на рис. 10.24, *г.*

Після того як система деякий час попрацює, пам'ять розділиться на декілька ділянок, частина з яких міститимуть сегменти, а частину – діри. Це явище назвають явищем шахівниці або ***зовнішньою фрагментацією***, що призводить до марної трати пам'яті на діри. З цим можна впоратися за рахунок ущільнення, показаного на рис. 10.24, *д*.

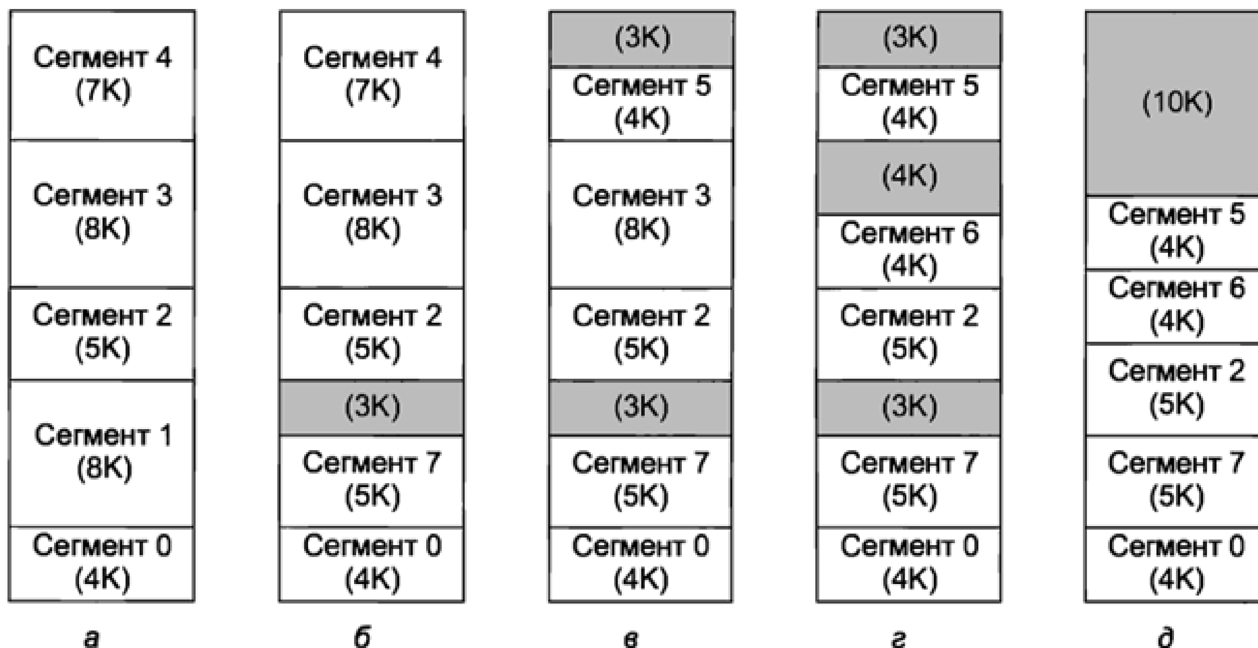


Рисунок 10.24 – Наростання зовнішньої фрагментації (*а – г*); позбавлення від зовнішньої фрагментації за рахунок ущільнення (*д*)

Оскільки таблиця сегментів має змінну довжину, залежну від розміру процесу, то найчастіше вона зберігається в оперативній пам'яті, а не в швидких регістрах. Використанням сегментів різного розміру цей спосіб схожий на динамічний розподіл пам'яті. Проте на відміну від динамічного розподілу в цьому випадку сегменти можуть займати декілька несуміжних розділів.

10.13 Сегментно-сторінкова організація пам'яті

І сторінкова, і сегментна організація мають свої переваги. Сторінкова організація усуває зовнішню фрагментацію, і тим самим забезпечує ефективне використання основної пам'яті. Крім того, оскільки переміщувані блоки мають фіксований, однаковий розмір, то полегшується створення ефективних алгоритмів управління пам'яттю.

Як видно з назви, цей метод є комбінацією сторінкового і сегментного розподілу пам'яті і, внаслідок цього, поєднує в собі переваги обох підходів.

При великому розмірі сегментів може стати незручним або навіть неможливим зберігання їх цілком в оперативній пам'яті. Це наштовхує на ідею застосування до них сторінкової організації, щоб мати справу тільки з тими сторінками, які потрібні в даний момент. Підтримка сторінкових сегментів реалізована в декількох системах.

10.13.1 Загальна схема організації сегментно-сторінкової пам'яті

Уперше сегментація зі сторінковою організацією пам'яті була застосована в ОС MULTICS (1965 р.) [12]. Система MULTICS забезпечувала кожну програму віртуальною пам'яттю розміром аж до 2^{18} сегментів (більше 250000), кожен з яких міг бути розміром до 65536 36-розрядних слів завдовжки. Розробники системи MULTICS вирішили трактувати кожен сегмент як віртуальну пам'ять і розбити його на сторінки, комбінуючи переваги сторінкової організації пам'яті з перевагою сегментації.

У такій комбінованій системі адресний простір користувача розбивається на ряд сегментів на розсуд програміста. Кожен сегмент у свою чергу розбивається на сторінки фіксованого розміру, які рівні сторінці фізичної пам'яті. З точки зору програміста, логічна адреса в цьому випадку складається з номера сегменту і зміщення в ньому. З позиції операційної системи зміщення в сегменті слід розглядати як номер сторінки певного сегменту і зміщення в ній (рис. 10.25).

Рисунок 10.25 – Сегментно-сторінкова організація пам'яті

Приклад отримання фізичної адреси і витяг з пам'яті необхідного елементу для цього способу представлений на рис. 10.26.

Цей спосіб організації віртуальної пам'яті вносить ще більшу затримку доступу до пам'яті. Необхідно спочатку вичислити адресу дескриптора сегменту і прочитати її, потім вичислити адресу елементу таблиці сторінок цього сегменту

і витягнути з пам'яті необхідний елемент, і вже тільки після цього можна до номера фізичної сторінки приписати номер комірки в сторінці (індекс). Щоб уникнути такої затримки, вводиться кешування, причому кеш, як правило, будується за асоціативним принципом.

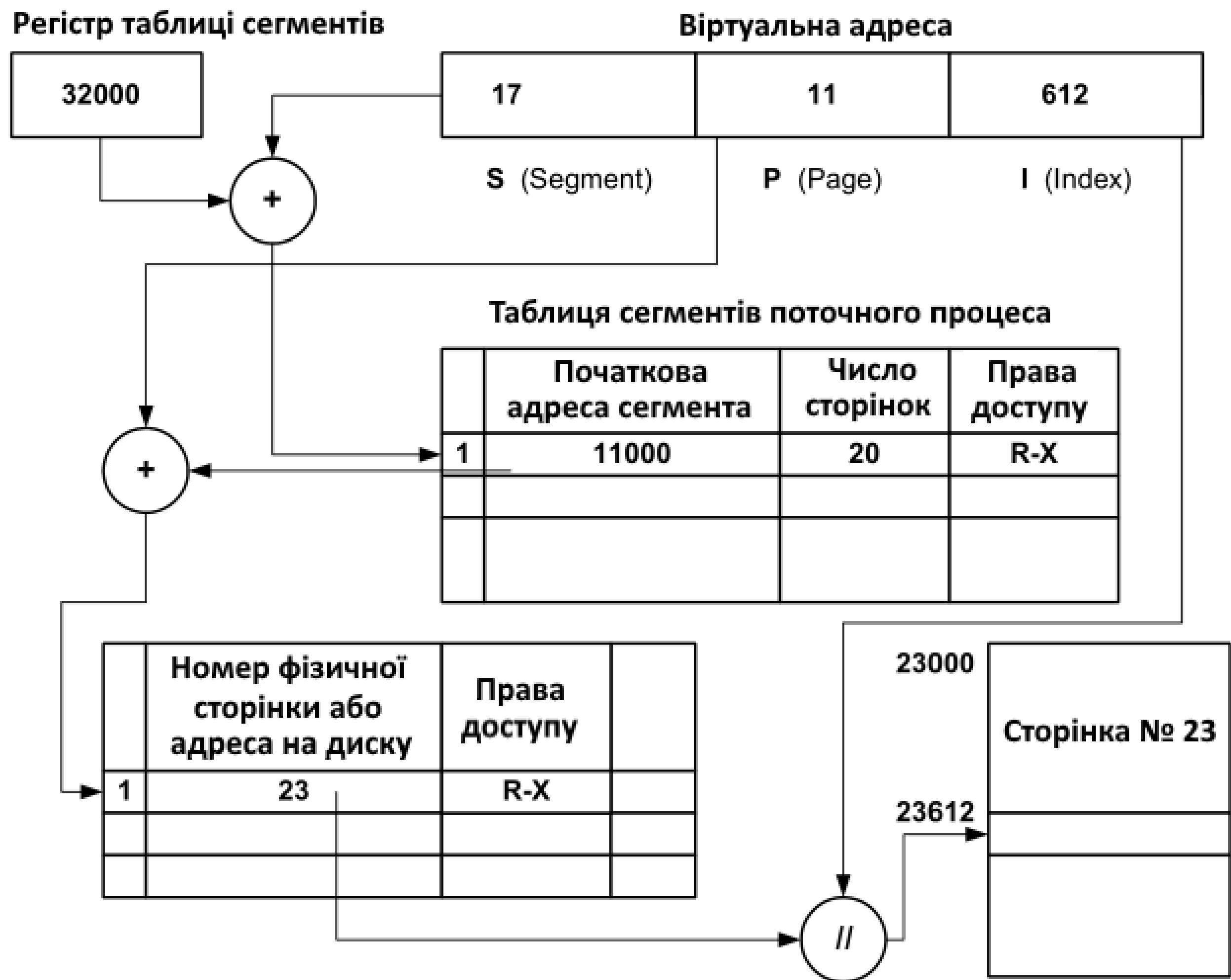


Рисунок 10.26 – Приклад отримання фізичної адреси при сегментно-сторінковій організації пам'яті

Апаратура системи MULTICS містила буфер швидкого перетворення адреси (TLB) розміром 16 слів, який був здатний здійснювати пошук паралельно по усіх своїх записах для заданого ключа. У буфері швидкого перетворення адреси зберігалися адреси 16 сторінок, до яких відбувалися самі останні звернення. Програми, в яких робочий набір був менше розміру TLB, зберігали адреси всього робочого набору в TLB, і отже, ці програми працювали ефективно, інакше відбувалася помилка TLB.

Сегментація зручна для реалізації захисту і спільного використання сегментів різними процесами. Оскільки кожен запис таблиці сегментів включає початкову адресу і значення довжини, програма не в змозі ненавмисно звернутися до основної пам'яті за межами сегменту. Для того щоб відрізнити сегменти від індивідуальних, записи таблиці сегментів, що розділяються, містять 1-бітове поле, що має два значення: *shared* (розділяється) або *private* (індивідуальний). Для здійснення спільного використання сегменту він

поміщається у віртуальний адресний простір декількох процесів, при цьому параметри відображення цього сегменту налаштовуються так, щоб вони відповідали одній і тій же області оперативної пам'яті.

Незважаючи на переваги над другими методами розподілу пам'яті сегментно-сторінкове розподіл пам'яті вимагає дуже значних витрат обчислювальних ресурсів і його не так просто реалізувати. Використовується він нечасто, причому в дорогих, потужних обчислювальних системах. Можливість реалізувати сегментно-сторінковий розподіл пам'яті закладена і в сімейство мікропроцесорів i80x86, однак внаслідок слабкої апаратної підтримки, труднощів при створенні систем програмування та операційної системи, практично він не використовується в ПК [9].

10.13.2 Сегментно-сторінкова організація пам'яті в Windows

Розглянемо стисло сучаснішу організацію сегментно-сторінкової пам'яті, яка застосовується в системі Intel Pentium. Докладніше сегментно-сторінкова організація пам'яті ОС Windows розглядалася в підрозділі «4.6 Засоби підтримки механізмів віртуальної пам'яті».

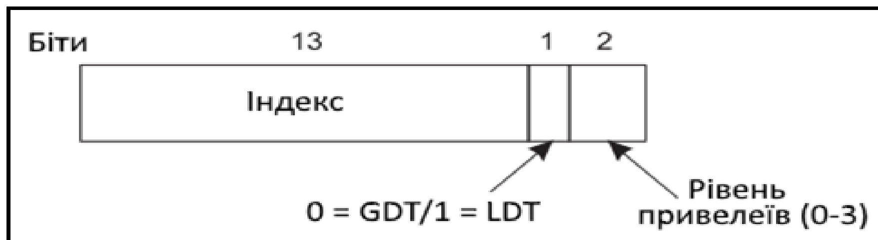
Віртуальна пам'ять в системі Pentium схожа на пам'ять в системі MULTICS, включаючи наявність як сегментації, так і сторінкової організації. Але система MULTICS має 256 Кб незалежних сегментів, кожен до 64 Кб 36-розрядних слів, а система Pentium підтримує 16 Кб незалежних сегментів, кожен до 1 млрд 32-розрядних слів. Хоча система Pentium має менше сегментів, їх більший розмір куди важливіше, оскільки програми, яким потрібно більш ніж 1000 сегментів, зустрічаються досить не часто, тоді як багатьом програмам потрібні великі за розміром сегменти.

Основа віртуальної пам'яті системи Pentium складається з двох таблиць: ***локальної таблиці дескрипторів*** – **LDT** (Local Descriptor Table) і ***глобальної таблиці дескрипторів*** – **GDT** (Global Descriptor Table). У кожної програми є своя власна таблиця LDT, але глобальна таблиця дескрипторів, яку спільно використовують усі програми в комп'ютері, всього одна. У таблиці LDT описуються сегменти, локальні для кожної програми, включаючи код цих програм, їх дані, стек тощо, а в таблиці GDT описуються системні сегменти, включаючи саму операційну систему.

Щоб отримати доступ до сегменту, програма, що працює в системі Pentium, спочатку завантажує селектор для цього сегменту в один з шести сегментних регістрів машини. Під час виконання програми регістр CS містить селектор для сегменту коду, а регістр DS зберігає селектор для сегменту даних. Кожен селектор, як показано на рис. 10.27, є 16-розрядним цілим числом, що складається з трьох полів:

- індексу, який задає послідовний номер дескриптора в таблиці GDT або LDT (13 біт);
- покажчика типу використовуваної таблиці дескрипторів: GDT або LDT (1 біт);

– необхідного рівня привілеїв – RPL (2 біти), це поле використовується механізмом захисту даних (0 – ядро, 1 – системні вузли, 2 – бібліотеки спільного доступу, 3 – призначені для користувача програми). Рівні привілейованості забороняють виконуваному коду звернутися до нижчого рівня.



Риунок. 10.27 – Формат селектора в Pentium

Дескриптор 0 заборонений. Його можна без всякого побоювання завантажити в сегментний регістр, щоб позначити, що цей сегментний регістр в даний момент недоступний. Спроба ним скористатися призведе до системного переривання. Щоб полегшити визначення місця розташування дескриптора, був майстерно підібраний формат селектора. Спочатку на основі біта 2 селектора вибирається локальна або глобальна таблиця дескрипторів. Потім селектор копіюється у внутрішній робочий регістр, і значення трьох молодших бітів встановлюються в 0. Нарешті, до цієї копії додається адреса однієї з таблиць, LDT або GDT, щоб отримати прямий покажчик на дескриптор. Наприклад, селектор 72 посилається на запис 9 (72/8) в глобальній таблиці дескрипторів, яка розташована за адресою в таблиці GDT+72.

Під час завантаження селектора в сегментний регістр, з локальної або глобальної таблиці дескрипторів витягається відповідний дескриптор, який, щоб прискорити звернення до нього, зберігається в мікропрограмних регістрах. Як показано на рис. 10.28, дескриптор сегмента складається з 8 байтів, в які входить базова адреса сегменту (32 біта), 20-бітний розмір сегмента в елементах (див. G) та інша інформація.

31<-- 32 біта -------------------------------------->0												Адрес
Base 15-0						Lim 15-0						0
Base 31-24	G	D	X	U	Lim19-16	P	DPL	S	Type	A	Base 23-16	4

Рисунок 10.28 – Формат дескриптора сегмента даних або коду в Pentium

Base – 32-бітова початкова адреса сегменту в лінійному фізичному адресному просторі.

Lim (Limit) – 20-бітовий розмір сегменту в елементах (см G) мінус 1.

G (*Granularity*) – біт гранулярності, якій задає величину сегментного елементу: 0 – байт, 1 – сторінка (4 Кб). Таким чином, максимальний розмір сегменту може складати 1 Mb байтів або 1 Mb сторінок (4 Gb).

P (*Present*) – біт присутності, що містить 1, якщо сегмент знаходиться у фізичній пам'яті.

DPL (*Descriptor Privilege Level*) – двубітне поле рівня привілеїв, що відповідає сегменту.

S (*System*) – ознака системного об'єкту (0 – системний).

Type – 3 біта, що визначають цільове використання сегменту (дані, код, стек, дозвіл на читання і запис).

A (*Accessed*) – біт доступу, що встановлюється при зверненні до сегменту.

D (*Default Size*) – біт розміру за умовчанням, який задає розмір операндів в сегменті: 0 – 16 бітів, 1 – 32 біта; у сегменті коду це відповідає 16-бітовому коду процесора 80286 і 32-бітовому Pentium.

X – зарезервований біт.

U (*User*) – призначений для використання програмістами, процесор його ігнорує.

Як тільки мікропрограма дізнається, який сегментний регістр використовується, вона може знайти у своїх внутрішніх регістрах повний дескриптор, що відповідає цьому селектору. Якщо сегмент не існує (селектор дорівнює 0) або в даний момент вивантажений, виникає системне переривання.

Потім апаратура використовує поле межі (*Розмір, Limit*), щоб перевірити, чи не виходить зміщення за межу сегменту, і в цьому випадку також виникає системне переривання. Для надання розміру сегменту в дескрипторі має бути 32-розрядне поле, але доступні тільки 20 біт, тому використовується інша схема. Якщо поле **G** (Granularity – міра деталізації) дорівнює нулю, в полі *Розмір (Limit)* міститься точний розмір сегменту аж до 1 Мб. Якщо воно дорівнює 1, то в полі *Розмір* надається розмір сегменту в сторінках, а не в байтах. У системі Pentium використовується фіксований розмір сторінок, рівний 4 Кб, тому 20 бітів достатні для сегментів розміром до 2^{32} байтів.

Припустимо, що сегмент знаходиться в пам'яті і зміщення потрапило в потрібний інтервал, тоді система Pentium додає 32-розрядне поле *База (Base)* в дескрипторі до зміщення, формуючи те, що називається ***лінійною адресою***, як показано на рис. 10.29.

Рисунок 10.29 – Перетворення пари селектор-зміщення в лінійний адрес

Поле *База* розбито на три частини, які розкидані по дескриптору для сумісності з процесором Intel 80286, в якому поле *База* має тільки 24 біта. По суті, поле *База* дозволяє кожному сегменту починатися в довільному місці всередині 32-розрядного лінійного адресного простору.

Якщо розбиття на сторінки відключене, то лінійна адреса інтерпретується як фізичнна адреса і відправляється в пам'ять для читання або запису. Якщо розбиття на сторінки використовується, то лінійна адреса розглядається як віртуальна і відображається у фізичну адресу з використанням таблиць сторінок. Таким чином, при відключеній сторінковій схемі пам' яті отримуємо чисту схему сегментації.

Також слід зазначити, що ця модель працює і в тому випадку, коли деякі додатки не вимагають сегментації, а просто задовольняються єдиним, розбитим на сторінки 32-розрядним адресним простором. Усі сегментні регістри можуть бути налагоджені тим же самим селектором, у дескрипторі якого поле *Base = 0, а поле Limit* встановлене на максимум. Тоді зміщення команди буде лінійною адресою і використовуватиметься тільки один адресний простір, що приведе до звичайної сторінкової організації пам' яті. Фактично таким чином працюють усі сучасні операційні системи для комп'ютерів x86.

З іншого боку, якщо включено підкачування сторінок, лінійна адреса інтерпретується як віртуальна і відображається на фізичну адресу за допомогою таблиці сторінок практично так само, як в попередніх прикладах. Єдине реальне утруднення полягає в тому, що при 32-розрядній віртуальній адресі і сторінці розміром 4 Кб сегмент може містити 1 млн сторінок (1 Мб = 2^{20} = 4Гб/4Кб), тому використовується дворівневе відображення з метою зменшення розміру таблиці сторінок для невеликих сегментів.

Віртуальна адреса як і раніше є парою: селектор, який визначає номер віртуального сегменту, і зміщення всередині цього сегменту. Перетворення віртуальної адреси виконується в два етапи: спочатку працює сегментний механізм, а потім результат його роботи поступає на вхід сторінкового механізму, який і обчислює шукану фізичну адресу.

Робота сегментного механізму в даному випадку багато в чому повторює його роботу при відключеному сторінковому механізмі. На підставі значення індексу в селекторі вибирається потрібний дескриптор з таблиці GDT або LDT. З дескриптора витягається базова адреса сегменту і складається зі зміщенням. Дескриптори і таблиці мають ту ж структуру. Проте є і принципова відмінність, яка полягає в інтерпретації утримуваного поля базової адреси в дескрипторах сегментів. Якщо раніше дескриптор сегменту містив базову адресу сегменту у фізичній пам'яті і при складанні цієї адреси зі зміщенням з віртуальної адреси виходила фізична адреса, то тепер дескриптор містить базову адресу сегменту у віртуальному адресному просторі, і в результаті його складання зі зміщенням отримуємо ***лінійний віртуальний адрес.***

Результуюча лінійна 32-розрядна віртуальна адреса передається сторінкового механізму для подальшого перетворення. Виходячи з того що розмір сторінки дорівнює 4 Кб (2^{12}), в адресі можна легко виділити номер віртуальної сторінки (старші 20 розрядів) і зміщення в сторінці (молодші 12 розрядів). Для відображення віртуальної сторінки у фізичну досить побудувати таблицю сторінок, кожен елемент якої містив би номер відповідної їй фізичної сторінки і її атрибути.

У процесорі Pentium так і зроблено, структура дескриптора сторінки показана на рис. 10.30. Крім номера сторінки (20 розрядів) дескриптор сторінки містить також поля, близькі за змістом відповідним полям дескриптора сегмента:

- **P** – біт присутності сторінки у фізичній пам'яті;
- **W** – біт дозволу запису в сторінку;
- **U** – біт – користувач/супервізор;
- **A** – ознака доступу, що мав місце, до сторінки;
- **D** – ознака модифікації вмісту сторінки;
- **AVL** – резерв для потреб операційної системи;
- **PWT** і **PCD** – біти, які управляють механізмом кешування сторінок.

20	3	2	1	1	1	1	1	1	1
№ сторінки	AVL	0	D	A	PCD	PWT	U	W	P

Рисунок 10.30 – Формат дескриптора сторінки

Таблиця сторінок може займати в пам'яті дуже значне місце – 4 байт*1 Мб = 4 Мб. Тому таблицю сторінок в силу її великого об'єму конструюють з двох рівнів: ***каталог таблиць сторінок (розділи)*** по 1024 дескриптори і ***таблиці сторінок***, що містить 1024 32-разрядних записів [28]. Один розділ займає одну сторінку (1024*4 байт – 4 Кб), тобто, 1024 розділи. Записи в каталозі вказують на таблицю сторінок, а записи в таблицях сторінок вказують на сторінкові блоки (рис. 10.31).

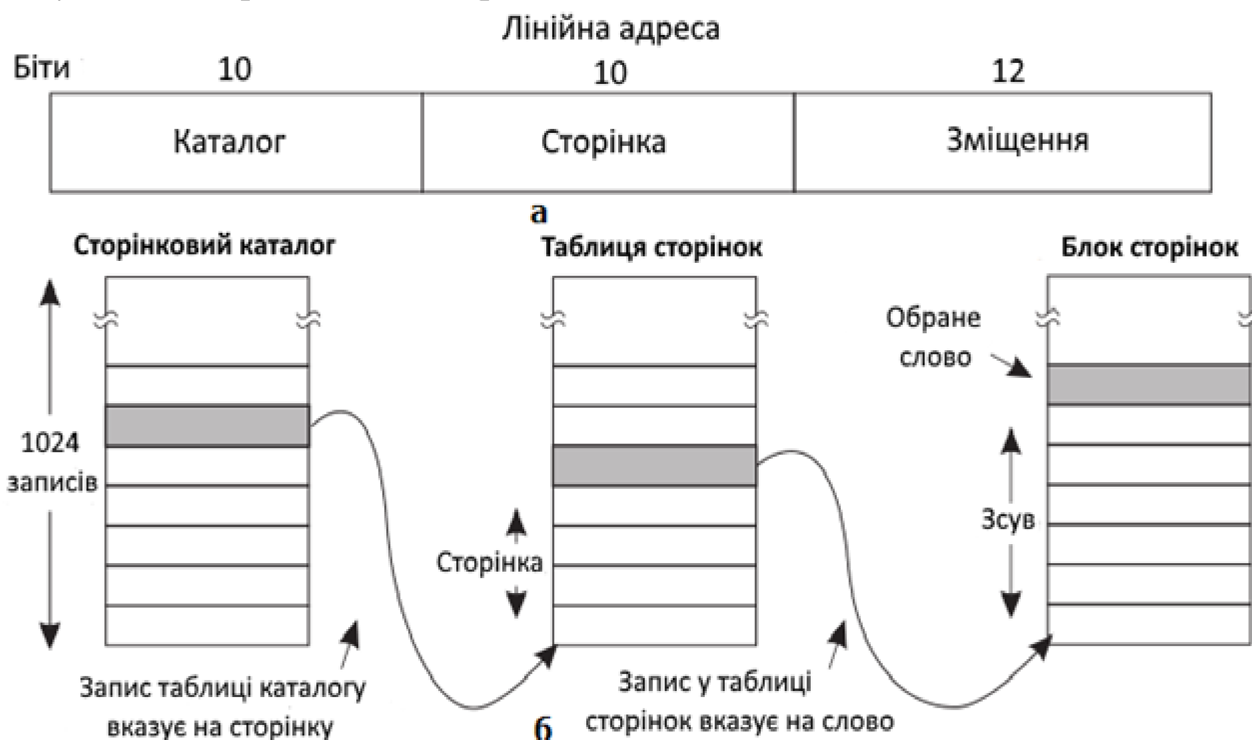


Рисунок 10.31 – Схема відображення лінійної адреси на фізичну

Лінійна адреса ділиться на три поля: *Каталог, Сторінка* і *Зміщення.* Поле *Каталог* використовується як індекс у сторінковому каталозі, що визначає

розташування покажчика на правильну таблицю сторінок. Поле *Сторінка* використовується як індекс в таблиці сторінок, щоб знайти фізичну адресу сторінкового блоку. Щоб отримати фізичну адресу, до адреси сторінкового блоку додається останнє поле *Зміщення*.

Поле номера віртуальної сторінки (старші 20 розрядів) ділиться на дві рівні частини по 10 розрядів – поле номера розділу і поле номера сторінки в розділі. На підставі заданого в регістрі CR3 номера фізичної сторінки, що зберігає таблицю розділів, і зміщення в цій сторінці, що задається полем номера розділу, процесор знаходить дескриптор віртуальної сторінки розділу.

Відповідно до атрибутів цього дескриптора визначаються права доступу до сторінки, а також наявність її у фізичній пам'яті. Якщо сторінки немає в оперативній пам'яті, то відбувається переривання, в результаті якого ОС повинна виконати завантаження необхідної сторінки в пам'ять.

Віртуальні сторінки таблиці сторінок, як і усі інші сторінки, можуть вивантажуватися на диск. Віртуальна сторінка, що зберігає таблицю розділів (сторінковий каталог), завжди знаходиться у фізичній пам'яті.

Кожен запис в таблиці сторінок має розмір 32 біти, 20 з яких містять номер сторінкового блоку. Інші біти включають біти доступу і біт зміни сторінки, що встановлюються апаратурою для операційної системи, біти захисту і інші службові біти (рис. 10.32) [28].

Рисунок 10.32 Структура рядка таблиці сторінок

Щоб уникнути повторних звернень до пам'яті, система x86, як і система MULTICS, має невеликий буфер швидкого перетворення адреси (TLB), який безпосередньо відображає найчастіші комбінації *Каталог – Сторінка* на фізичну адресу сторінкового блоку. Механізм, показаний на рис. 10.31, задіюється лише за відсутності поточної комбінації в буфері TLB, при цьому сам буфер оновлюється. Якщо відсутність потрібної інформації в буфері TLB зустрічається досить не часто, то система досягає непоганої продуктивності [9].

Перетворення лінійної віртуальної адреси у фізичну відбувається таким чином (рис. 10.33).

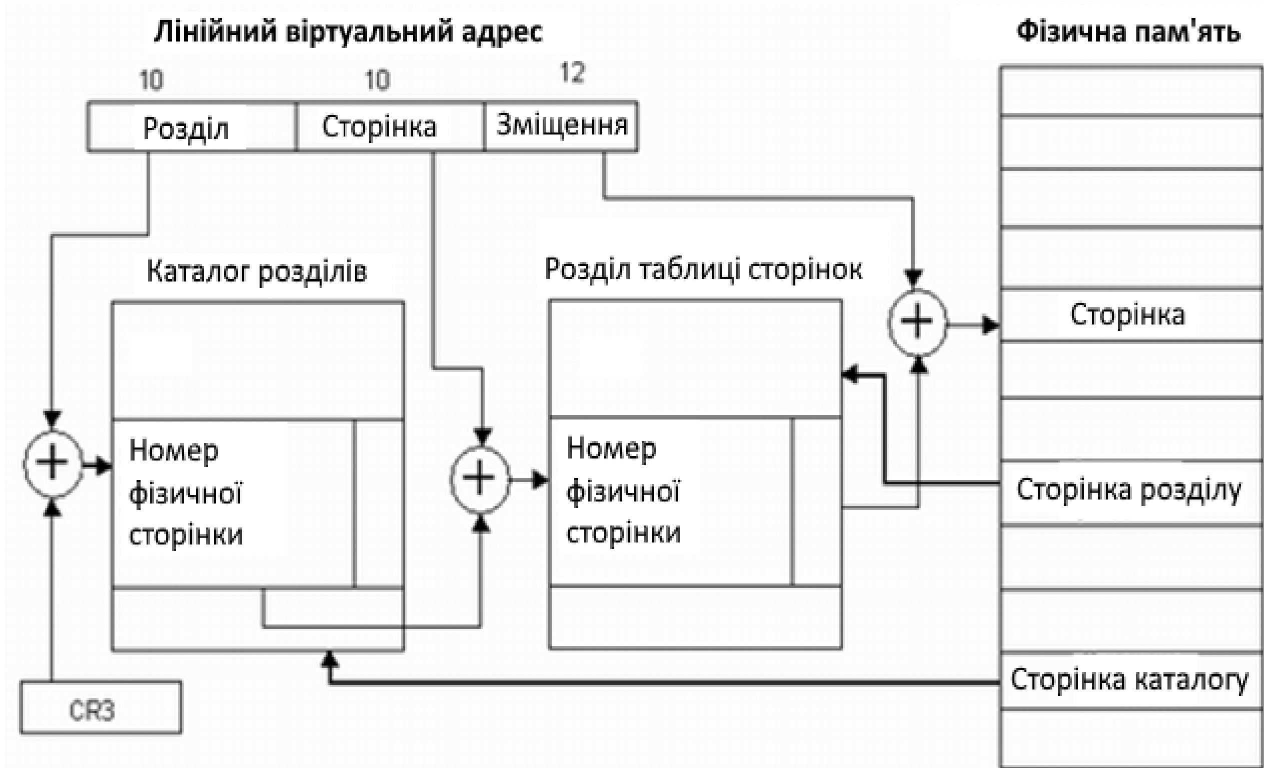


Рисунок 10.33 – Перетворення віртуального адреса у фізичну адресу

Сегментно-сторінкова організація пам'яті у Windows показана на рис. 10.34.

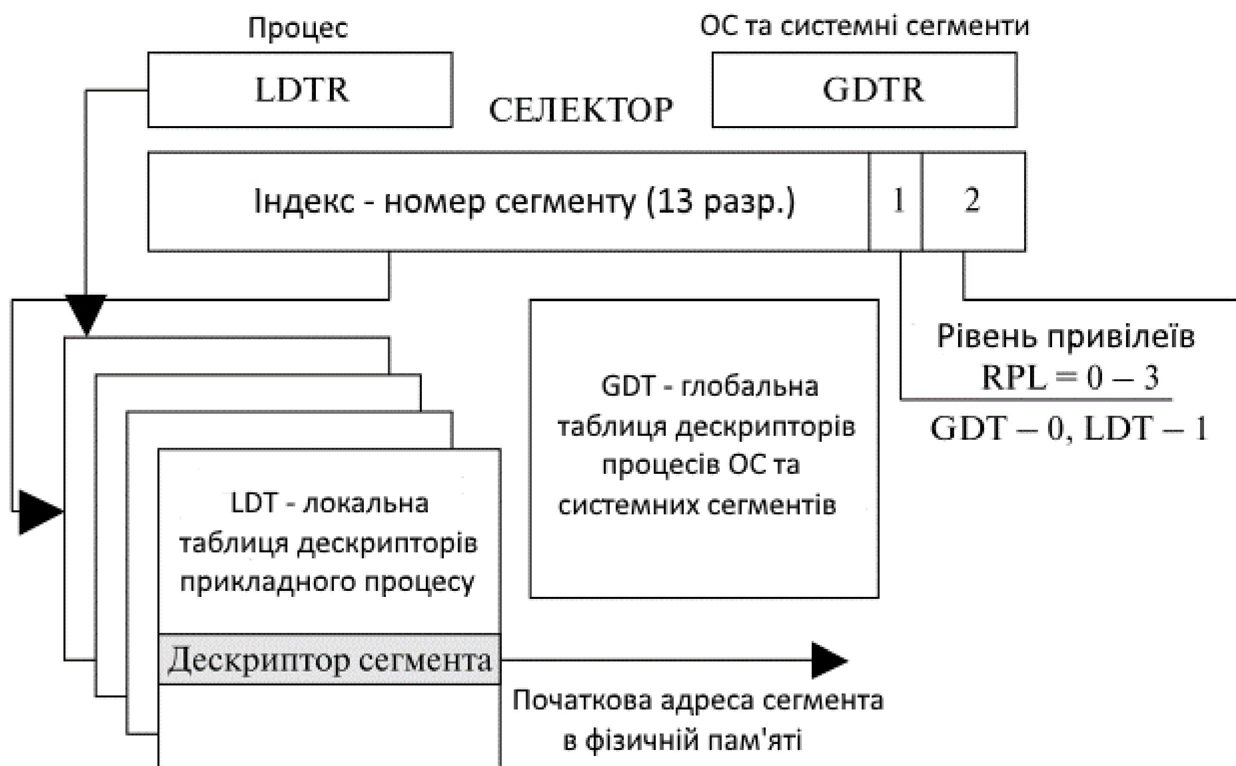


Рисунок 10.34 – Сегментно-сторінкова організація пам'яті у Windows

10.14 Свопінг

Різновидом віртуальної пам'яті є ***свопінг*** (swapping) або ***звичайне підкачування.*** Необхідною умовою для виконання задачі є завантаження її в оперативну пам'ять, об'єм якої обмежений. Для підвищення завантаження процесора був запропонований метод організації обчислювального процесу, що називається ***свопінгом*** (рис. 10.35). Процеси, що знаходяться в стані очікування, тимчасово цілком вивантажуються на диск. Планувальник ОС не виключає їх зі свого розгляду, і при настанні умов активізації деякого процесу цей процес переміщається в оперативну пам'ять. Якщо вільного місця в оперативній пам'яті бракує, то вивантажується інший процес. У разі переривання роботи процесу він переміщається назад у зовнішню пам'ять.

Якщо на комп'ютері одночасно виконується велике число обчислювальних задач і задач з інтенсивним введенням/виведенням, то вдається добитися високої ефективності використання центрального процесора.

Свопінг є окремим випадком віртуальної пам'яті і, отже, простіший в реалізації спосіб спільного використання оперативної пам'яті і диска. Проте підкачуванню властива надмірність. Коли ОС вирішує активізувати процес, для його виконання, як правило, не вимагається завантажувати в оперативну пам'ять усі його сегменти повністю. Крім того, системи, що підтримують свопінг, мають ще один дуже суттєвий недолік: вони не здатні завантажити для виконання процес, віртуальний адресний простір якого перевищує наявну вільну пам'ять.

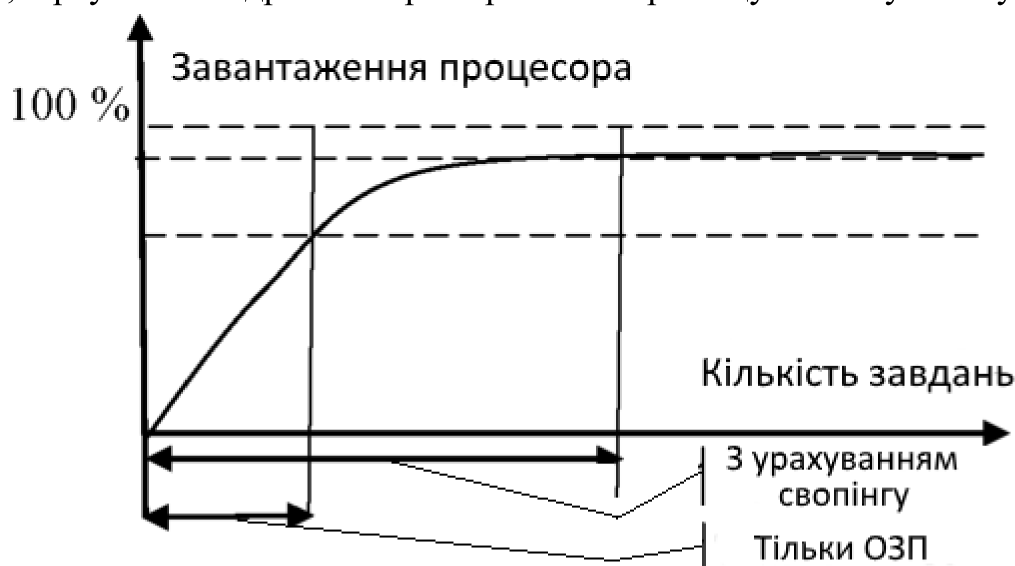


Рисунок 10.35 – Завантаження процесора при використанні свопингу

Враховуючи ці недоліки «чистого» свопінгу, особливо у великих втратах часу на завантаження або вивантаження процесів, в сучасних операційних системах використовуються модифіковані варіанти свопінгу. Так, наприклад, у багатьох версіях операційної системи UNIX свопінг включається тільки в тому випадку, коли кількість процесів у пам'яті стає занадто великою.

Контрольні питання і тести до розділу 10

Контрольні питання

1. Які основні функції ОС з управління пам'яттю?
2. Яка фрагментація властива методам розподілу пам'яті фіксованими розділами?
3. Які задачі ставляться перед ОС при реалізації динамічного методу управління пам'яттю?
4. Яка фрагментація властива методам динамічного розподілу пам'яті?
5. Які методи боротьби з фрагментацією застосовуються при реалізації динамічного методу управління пам'яттю?
6. Як фіксований, так і динамічний розподіл пам'яті мають переваги і недоліки. Яка система є компромісною в цьому плані?
7. У якій сучасній ОС використовується модифікована версія системи двійників для розподілу пам'яті?
8. Дайте поняття віртуальної пам'яті.
9. У якій ОС вперше була використана віртуальна пам'ять?
10. Що стверджує правило локалізації (локальності)?
11. Наведіть приклади просторової і часової локальності.
12. Назвіть найпоширеніші методи організації віртуальної пам'яті.
13. Як називаються сторінки, на які ділиться віртуальний адресний простір процесу і уся оперативна пам'ять комп'ютера?
14. Як називається частина процесу, яка розташована в деякий момент часу в основній пам'яті?
15. Яку інформаційну структуру створює ОС для кожного процесу при сторінковій організації пам'яті?
16. Яка інформація зберігається в кожному записі таблиці (дескриптор сторінки)?
17. Щоб обійти проблему необхідності постійного зберігання в пам'яті величезних таблиць сторінок, деякі процесори використовують дворівневу таблицю сторінок. Які структури даних використовуються при такій схемі доступу до фізичної сторінки пам'яті?
18. Назвіть синоніми високошвидкісного кешу для записів таблиць сторінок.
19. Чому пам'ять високошвидкісного кешу називається асоціативною?
20. Для чого застосовуються інвертовані таблиці сторінок?
21. Яка інформація зберігається в кожному записі інвертованої таблиці сторінок?
22. Яка фрагментація властива сторінковій організації віртуальної пам'яті?
23. Назвіть переваги сегментної організації віртуальної пам'яті перед сторінковою.
24. Яка фрагментація властива сегментній організації віртуальної пам'яті?
25. Назвіть недоліки сегментного розподілу пам'яті.
26. В якій ОС була вперше застосована сегментація з сторінковою організацією пам'яті?

Тести

1. Свопінгом сегментів називається переміщення:
   1) блоків файлу між каталогами файлової системи;
   2) блоків даних між процесом і ядром операційної системи;
   3) сегментів даних між стеком і оперативною пам'яттю;
   4) сегментів між оперативною і зовнішньою пам'яттю.
2. Якщо розмір програми істотно більше об'єму доступної оперативної пам'яті, то використання віртуальної пам'яті в однопрограмному режимі призводить до . . . процесу.
   1) аварійного завершення;
   2) прискорення;
   3) перезапуску;
   4) уповільнення виконання.
3. Віртуальна пам'ять дозволяє:
   1) відмовитися від надання процесам оперативної пам'яті;
   2) завантажувати програми, скомпільовані для іншого процесора;
   3) завантажувати певну кількість програм, розмір яких перевищує об'єм доступної фізичної пам'яті.
4. Сегментна організація пам'яті ... окремо скомпільованих процедур.
   1) складається з;
   2) спрощує компонування;
   3) неможлива без;
   4) ускладнює компонування.
5. При сторінковій організації пам'яті таблиця сторінок може розміщуватися:
   1) у спеціальній швидкій пам'яті процесора і в оперативній пам'яті;
   2) тільки в процесорі;
   3) тільки в оперативній пам'яті;
   4) в оперативній пам'яті і на диску.
6. Сторінкова організація пам'яті призначена для:
   1) отримання великого адресного простору без придбання додаткової фізичної пам'яті;
   2) полегшення спільного використання процедур, бібліотек і масивів даних;
   3) підвищення рівня захисту програм і даних;
   4) логічного розділення програм і даних.
7. Що таке таблиця сторінок процесу?
   1) структура, що організована для контролю доступу до сторінок процесу;
   2) структура, що організована для обліку вільних і зайнятих сторінкових блоків;
   3) структура, що використовується для відображення логічного адресного простору у фізичний при сторінковій організації пам'яті;
   4) структура, що створена для доступу до файлів процесу на диску.
8. Максимальний розмір логічного адресного простору визначається:
   1) об'ємом оперативної пам'яті;

2) розрядністю процесора;
3) об'ємом жорсткого диска;
4) об'ємом фізичної пам'яті.

9. Вичисліть номер сторінки і зміщення для логічної адреси 32768, якщо розмір сторінки рівний 4Кб. Сторінки нумеруються, починаючи з 0:
   1. 3 і 2048;
   2. 5 і 4096;
   3. 6 і 512;
   4. 7 і 0.

10. Найефективнішим способом управління оперативною пам'яттю є:
    1) розподіл пам'яті переміщуваними розділами;
    2) розподіл пам'яті динамічними розділами;
    3) віртуальна пам'ять;
    4) розподіл пам'яті фіксованими розділами.

11. Аномалія Беледи полягає в тому, що:
    1) певні послідовності звернень до сторінок призводять до зменшення числа сторінкових порушень при збільшенні кадрів, виділених процесу;
    2) будь-які послідовності звернень до сторінок призводять до зменшення числа сторінкових порушень при збільшенні кадрів, виділених процесу;
    3) деякі послідовності звернень до сторінок призводять до збільшення числа сторінкових порушень при зменшенні кадрів, виділених процесу;
    4) деякі послідовності звернень до сторінок призводять до збільшення числа сторінкових порушень при збільшенні виділених процесу кадрів.

12. Інвертована таблиця сторінок дає можливість:
    1) отримати номер сторінкового кадру за номером віртуальної сторінки;
    2) прискорити процес трансляції адреси;
    3) зменшити об'єм пам'яті, що витрачається на відображення віртуального адресного простору у фізичний.

13. Скільки записів у таблиці сторінок в системі з 32-розрядною архітектурою і розміром сторінки 4Кб?
    1) 2^{24};
    2) 2^{20};
    3) 2^{22};
    4) 2^{16}.

14. Таблиця сторінок процесу – це:
    1) структура, яка використовується для відображення логічного адресного простору у фізичний при сторінковій організації пам'яті;
    2) структура, яка організована для обліку вільних і зайнятих сторінкових блоків;

3) структура, що організована для контролю доступу до сторінок процесу.

15. Що розуміється під терміном «зовнішня фрагментація»?
    1) втрата частини пам'яті в схемі зі змінними розділами;
    2) втрата частини пам'яті, яка виділена процесу;
    3) втрата частини пам'яті, яка не виділена жодному процесу;
    4) наявність фрагментів пам'яті, зовнішніх стосовно процесу.
16. Що таке «віртуальна пам'ять»?
    1) спосіб розділення адресного простору між задачами;
    2) спосіб розділення адресного простору між ядром і іншими задачами;
    3) спосіб доступу до фізичної пам'яті комп'ютера, при якому задачі можуть дозволяти або забороняти доступ інших задач до областей пам'яті, що належить їм;
    4) сукупність програмно-апаратних засобів, що дозволяють користувачам писати програми, розмір яких перевершує наявну оперативну пам'ять.
17. Виберіть алгоритм розподілу пам'яті, який не передбачає використання зовнішньої пам'яті:
    1) сегментний розподіл;
    2) сегментно-сторінковий розподіл;
    3) сторінковий розподіл;
    4) динамічними розділами.
18. Пам'ять, що розділяється, – це:
    1) сегмент віртуальної пам'яті, відображений у фізичний адресний простір декількох процесів;
    2) сегмент віртуальної пам'яті, відображений у віртуальний адресний простір декількох процесів;
    3) сегмент фізичної пам'яті, відображений у віртуальний адресний простір декількох процесів.
19. При активації процесу базові адреси його таблиці сегментів і таблиці сторінок завантажуються в:
    1) оперативну пам'ять;
    2) стек;
    3) спеціальні регістри процесора;
    4) дескриптор процесу.
20. Таблиця сторінок процесу – це:
    1) структура, що організована для контролю доступу до сторінок процесу;
    2) структура, яка використовується для відображення логічного (віртуального) адресного простору у фізичний при сторінковій організації пам'яті;
    3) структура, що організована для обліку вільних і зайнятих сторінкових блоків.
21. Із-за непередбачуваності розмірів сегментів в оперативній пам'яті сегментний розподіл пам'яті схильний до:

1) зовнішньої фрагментації;
2) внутрішньої фрагментації;
3) порушення захисту;
4) перевантаження.

22. Основним недоліком методу розподілу пам'яті переміщуваними розділами є:
    1) дефрагментаци;
    2) фрагментація;
    3) низька ефективність використання пам'яті;
    4) значна кількість часу, що витрачається на процедуру стискування.
23. При використанні схеми сегментно-сторінкового розподілу пам'яті обмін між диском і оперативною пам'яттю здійснюється:
    1) сторінками фіксованого розміру;
    2) сегментами фіксованого розміру;
    3) сегментами змінного розміру;
    4) сторінками змінного розміру.
24. Для свопінгу характерно наступне:
    1) між оперативною пам'яттю і диском переміщаються частини образів процесів;
    2) образи процесів вивантажуються на диск і повертаються в оперативну пам'ять цілком;
    3) на диск вивантажуються усі таблиці сторінок 2-го рівня і повертаються в оперативну пам'ять на вимогу.
25. Яка з схем управління пам'яттю схильна до внутрішньої фрагментації?
    1) схема з динамічними розділами;
    2) сторінкова організація;
    3) сегментна організація.
26. Що розуміється під терміном «Зовнішня фрагментація»?
    1) втрата частини пам'яті, не виділеної жодному процесу;
    2) втрата частини пам'яті, виділеної одному процесу;
    3) наявність фрагментів пам'яті, зовнішніх стосовно процесу;
    4) втрата частини пам'яті в схемі зі змінними розділами.
27. У системі із сторінково-сегментною організацією пам'яті кожному процесу виділяють 64 Кб адресного простору для трьох сегментів процесу: код розміром 32 Кб, сегменту даних розміром 16400 байт, стеку – 15800 байт. Чи досить адресного простіру процесу для розміщення цих сегментів, якщо розмір сторінки дорівнює: 2 Кб, 1 Кб, 512 байт?
    1) так, якщо розмір сторінки дорівнює 2 Кб;
    2) так, якщо розмір сторінки дорівнює 1 Кб;
    3) так, якщо розмір сторінки дорівнює 512 байт;
    4) не один з перечислених розмірів сторінок не підходить.
28. Допустимо, що розмір сторінки пам'яті представляє 4 Кб. Кожен елемент таблиці сторінок (дескриптор) займає 4 байти, кожна таблиця сторінок повинна вміщуватися на одній сторінці. Скільки рівнів таблиць сторінок буде потрібно, щоб адресувати 32-бітний адресний простір?

1) один рівень;
2) чотири рівні;
3) три рівні;
4) два рівні.

29. У спеціальному високошвидкісному кеші для записів сторінок таблиці, який називають **буфером швидкого перетворення адреси,** або **буфером пошуку трансляції (translation TLB асоціативною пам'яттю,** пошук номера віртуальної сторінки виконується як:
    1) лінійний пошук;
    2) індексний пошук (номер логічної сторінки використовується в якості індекса);
    3) порівняння адреси з усіма записами кеша одночасно;
    4) бінарний пошук.

30. Нехай середній розмір процесу займає S байт, а сторінки – P байт, запис для кожної сторінки таблиці – R байт. Тоді приблизна кількість сторінок, необхідна для процесу, рівна S/P, що займе S*R/P байт для таблиці сторінок. Втрата пам'яті в останній сторінці процесу внаслідок внутрішньої фрагментації, рівна P/2. Загальні накладні витрати (C) внаслідок підтримки таблиці сторінок і втрати від внутрішньої фрагментації складають: C = S*R/P + P/2.

    Знайдіть **оптимальний розмір сторінки** (P) для середнього розміру процесу S = 1 Мб, розміру запису в таблиці сторінок R = 8 байт:
    1) 512 байт;
    2) 1 Кб;
    3) 2 Кб;
    4) 4 Кб.

31. У системі зі сторінково-сегментною організацією пам'яті кожному процесу виділяють 32 Кб адресного простору для трьох сегментів процесу: код розміром 16 Кб, сегменту даних розміром 9200 байт, стеку – 7100 байт. Чи достатньо адресного простору процесу для розміщення цих сегментів, якщо розмір сторінки дорівнює: 2 Кб, 1 Кб, 512 байт?
    1) так, якщо розмір сторінки дорівнює 2 Кб;
    2) так, якщо розмір сторінки дорівнює 1 Кб;
    3) так, якщо розмір сторінки дорівнює 512 байт;
    4) ні один із перечислених розмірів сторінок не підходить.

32. В якому методі організації розподілу пам'яті використовується процедура ущільнення процесів:
    1) динамічний розподіл пам'яті переміщуваними розділами;
    2) сторінкова організація віртуальної пам'яті;
    3) розподіл пам'яті фіксованими розділами;
    4) розподіл пам'яті динамічними розділами.

11 УПРАВЛІННЯ СТОРІНКОВОЮ ПАМ'ЯТТЮ

Одна з основних задач ОС – управління віртуальною пам'яттю. При виборі стратегії розв'язання цієї задачі ключовим питанням стає продуктивність: вимагається скоротити кількість переривань через відсутність сторінки в основній пам'яті, оскільки їх обробка призводить до істотних накладних витрат. Крім того, ОС повинна активізувати готовий до роботи процес на час виконання повільних операцій введення-виведення.

Що ж відбувається, коли потрібної сторінки в пам'яті немає або операція звернення до пам'яті недопустима? Природно, що операційна система має бути якось сповіщена про те, що сталося. Для цього використовується механізм виняткових ситуацій. При спробі виконати подібне звернення до віртуальної сторінки виникає виняткова ситуація «***сторінкове порушення» (*****page fault**), що призводить до виклику спеціальної послідовності команд для обробки конкретного виду сторінкового порушення.

Сторінкове порушення може відбуватися в найрізноманітніших випадках: за відсутності сторінки в оперативній пам'яті, при спробі запису в сторінку з атрибутом «тільки читання» або при спробі читання або запису сторінки з атрибутом «тільки виконання». У будь-якому з цих випадків викликається обробник сторінкового порушення, що є частиною операційної системи. Йому передається причина виникнення виняткової ситуації і віртуальна адреса, звернення до якої викликало порушення.

Нас цікавитиме конкретний варіант сторінкового порушення – звернення до відсутньої сторінки, оскільки саме його обробка багато в чому визначає продуктивність сторінкової системи. Коли програма звертається до віртуальної сторінки, відсутньої в основній пам'яті, операційна система повинна виділити сторінку основної пам'яті, перемістити в неї копію віртуальної сторінки із зовнішньої пам'яті і модифікувати відповідний елемент таблиці сторінок.

11.1 Стратегії управління сторінковою пам'яттю

Підвищення продуктивності обчислювальної системи може бути досягнуте за рахунок зменшення частоти сторінкових порушень, а також за рахунок збільшення швидкості їх обробки. Час ефективного доступу до відсутньої в оперативній пам'яті сторінки складається з:

1. Обслуговування виняткової ситуації (page fault).
2. Читання (підкачування) сторінки з вторинної пам'яті. Іноді, при нестачі місця в основній пам'яті, необхідно виштовхнути одну із сторінок з основної пам'яті у вторинну, тобто здійснити заміщення сторінки.
3. Відновлення виконання процесу, що викликав даний page fault.

Для розв'язання першої і третьої задач ОС виконує до декількох сотень машинних інструкцій впродовж декількох десятків мікросекунд. Час підкачування сторінки близький до декількох десятків мілісекунд. Проведені дослідження показують, що ймовірності page fault $5x10^{-7}$ виявляється достатньо, щоб понизити продуктивність сторінкової схеми управління пам'яттю на 10%.

Таким чином, зменшення частоти page faults є одним з ключових задач системи управління пам'яттю. Його розв'язання пов'язане з правильним вибором алгоритму заміщення сторінок. Тому програмне забезпечення управління пам'яттю ОС має бути пов'язане з реалізацією нижченаведених стратегій.

Стратегія вибірки визначає, в який момент слід переписати сторінку з вторинної пам'яті в первинну. Існує два основні варіанти вибірки – за запитом і з упередженням. Алгоритм вибірки за запитом вступає в дію в той момент, коли процес звертається до відсутньої сторінки, вміст якої знаходиться на диску. Його реалізація полягає в завантаженні сторінки з диска у вільну фізичну сторінку і корекції відповідного запису таблиці сторінок.

Алгоритм вибірки з упередженням здійснює випереджаюче читання, тобто окрім сторінки, що викликала виняткову ситуацію, в пам'ять завантажується декілька сторінок, що її оточують (зазвичай сусідні сторінки розташовуються в зовнішній пам'яті послідовно і можуть бути зчитані за одне звернення до диска). Такий алгоритм покликаний зменшити накладні витрати, пов'язані з великою кількістю виняткових ситуацій, що виникають при роботі зі значними об'ємами даних або коду. Крім того, оптимізується робота з диском.

Стратегія розміщення визначає, в яку ділянку первинної пам'яті помістити сторінку, що поступає. У системах із сторінковою організацією все просто – у будь-який вільний сторінковий кадр. У разі систем з сегментною організацією потрібна стратегія, аналогічна стратегії з динамічним розподілом.

Стратегія заміщення визначає, яку сторінку треба виштовхнути в зовнішню пам'ять, щоб звільнити місце в оперативній пам'яті. Розумна стратегія заміщення, реалізована у відповідному алгоритмі заміщення сторінок, дозволяє зберігати в пам'яті найнеобхіднішу інформацію і тим самим понизити частоту сторінкових порушень. Заміщення повинне відбуватися з урахуванням виділеної кожному процесу кількості кадрів. Крім того, треба вирішити, чи повинна сторінка, що заміщається, належати процесу, який ініціював заміщення, чи вона має бути вибрана серед усіх кадрів основної пам'яті.

11.2 Основні алгоритми заміщення сторінок

Розмір віртуальної пам'яті для кожного процесу може істотно перевершувати розмір основної пам'яті. Це означає, що при виділенні сторінки основній пам'яті з великою ймовірністю не вдасться знайти вільний сторінковий кадр. У цьому випадку операційна система відповідно до закладених в неї критеріїв повинна:

- знайти деяку зайняту сторінку основної пам'яті;
- перемістити в разі потреби її вміст в зовнішню пам'ять;
- переписати в цей сторінковий кадр вміст потрібної віртуальної сторінки із зовнішньої пам'яті;
- належним чином модифікувати необхідний елемент відповідної таблиці сторінок;
- продовжити виконання процесу, якому ця віртуальна сторінка знадобилася.

Відмітимо, що при заміщенні доводиться двічі передавати сторінку між основною і вторинною пам'яттю. Процес заміщення може бути оптимізований за рахунок використання біта модифікації (один з атрибутів сторінки в таблиці сторінок). Біт модифікації встановлюється комп'ютером, якщо хоч би один байт був записаний на сторінку. При виборі кандидата на заміщення перевіряється біт модифікації. Якщо біт не встановлений, немає необхідності переписувати цю сторінку на диск, її копія на диску вже є. Подібний метод також застосовується до read-only-сторінок, вони ніколи не модифікуються. Ця схема зменшує час обробки page fault.

Існує велика кількість різноманітних алгоритмів заміщення сторінок. Усі вони діляться на ***локальні*** і ***глобальні***. Локальні алгоритми заміщення сторінок, на відміну від глобальних, виділяють для заміщення з фізичної пам'яті одну з його сторінок, а не сторінки інших процесів. Глобальний же алгоритм заміщення в разі виникнення виняткової ситуації задовольниться звільненням будь-якої фізичної сторінки, незалежно від того, якому процесу вона належала.

Глобальні алгоритми мають ряд недоліків. По-перше, вони роблять одні процеси чутливими до поведінки інших процесів. Наприклад, якщо один процес у системі одночасно використовує велику кількість сторінок пам'яті, то усі інші процеси в результаті відчуватимуть сильне уповільнення через нестачу кадрів пам'яті для своєї роботи. По-друге, некоректно працюючий процес може підірвати роботу всієї системи, намагаючись захопити більше пам'яті. Тому в багатозадачній системі використовують складніші локальні алгоритми.

Застосування локальних алгоритмів вимагає зберігання в операційній системі списку фізичних кадрів, виділених кожному процесу. Цей список сторінок іноді називають ***резидентною множиною*** процесу.

Ефективність алгоритму оцінюється на конкретній послідовності посилань до пам'яті, для якої підраховується число ***page faults.*** Ця послідовність називається рядком звернень (reference string). Ми можемо генерувати рядок звернень штучним чином за допомогою датчика випадкових чисел або трасуючи конкретну систему.

Більшість процесорів мають прості апаратні засоби, що дозволяють збирати деяку статистику звернень до пам'яті. Ці засоби включають два спеціальні прапори на кожен елемент таблиці сторінок. Прапор посилання (reference біт) автоматично встановлюється, коли відбувається будь-яке звернення до цієї сторінки, а прапор зміни (modify біт) встановлюється, якщо робиться запис в цю сторінку. Операційна система періодично перевіряє установку таких прапорів, для того щоб виділити активно використовувані сторінки, після чого значення цих прапорів скидаються.

Незалежно від стратегії управління резидентною множиною є ряд основних алгоритмів для вибору сторінки на заміщення:

- оптимальний алгоритм (OPT);
- алгоритм «першим увійшов – першим вийшов» (FIFO);
- алгоритм виштовхування сторінки, яка найдовше не використовувалася;
- годиннковий і модифікований годинниковий алгоритми.

11.3 Оптимальний алгоритм

Оптимальна алгоритм полягає у виборі для заміщення тієї сторінки, звернення до якої буде через найбільший проміжок часу в порівнянні з усіма іншими сторінками. Як буде показано далі, цей алгоритм призводить до мінімальної кількості переривань через відсутність сторінки. Зрозуміло, що реалізувати такий алгоритм неможливо, оскільки для цього системі потрібно знати всі майбутні події. ОС не знає, до якої сторінки буде наступне звернення. Однак цей алгоритм є стандартом якості, з яким порівнюються реальні алгоритми.

На рис. 11.1 наведено приклад оптимальної стратегії. Передбачається, що для даного процесу використовується фіксований розподіл кадрів (фіксований розмір резидентної множини, що складається з трьох кадрів). Виконання процесу призводить до звернення п'яти різних сторінок. Оптимальна стратегія призводить після заповнення всієї множини кадрів до трьох переривань (звернення до сторінок), які позначені на малюнку буквами **F**.

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Оптимальний алгоритм	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
3 page faults					**F**		**F**			**F**		

Рисунок 11.1 – Приклад роботи оптимального алгоритму

11.3.1 Першим увійшов – першим вийшов

Стратегія «**першим увійшов – першим вийшов**» (**FIFO**, виштовхування першої сторінки, що надійшла) розглядає кадри сторінок процесу як циклічний буфер, з циклічним видаленням сторінок. Все, що потрібно для реалізації цього алгоритму, – це покажчик, який циклічно проходить по кадрах сторінок процесу.

Це одна з найпростіших в реалізації стратегій заміщення. Логіка її роботи полягає в тому, що заміщається сторінка, яка перебуває в основний пам'яті довше інших. Однак далеко не завжди ця сторінка використовується не часто; дуже часто деяка область даних або коду інтенсивно використовується програмою, і сторінки з цієї області при використанні описаної стратегії будуть завантажуватися і розвантажуватися. Робота алгоритму проілюстрована на рис. 11.2.

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Алгоритм FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
6 page faults					**F**	**F**	**F**		**F**		**F**	**F**

Рисунок 11.2 – Приклад роботи алгоритму FIFO

Стратегія «першим увійшов – першим вийшов» реалізується дуже просто, але відносно не часто призводить до кращих результатів. Багато з алгоритмів FIFO є варіантами схеми, відомої як годинникова стратегія (clock policy).

На перший погляд здається очевидним, що чим більше в пам'яті сторінкових кадрів, тим рідше будуть мати місце page faults. Але це не завжди так. Як встановив Біледі (Belady), певні послідовності звернень до сторінок в дійсності призводять до збільшення числа сторінкових порушень при збільшенні кадрів, виділених процесу. Це явище носить назву «***Аномалії Біледі»*** або «аномалії FIFO» (рис. 11.3) [12].

Черга сторінок	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**
Найстаріша сторінка	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Найновіша сторінка			0	1	2	3	0	0	0	1	4	4
(*a*) ***9 page faults***	**F**	**F**	**F**	**F**	**F**	**F**	**F**			**F**	**F**	
Черга сторінок	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**
Найстаріша сторінка	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Найновіша сторінка				0	0	0	1	2	3	4	0	1
(*b*) ***10 page faults***	**F**	**F**	**F**	**F**			**F**	**F**	**F**	**F**	**F**	**F**

Рисунок 11.3 – Аномалія Біледі: (*a*) – FIFO з трьома сторінковими кадрами; (*b*) – FIFO з чотирма сторінковими кадрами

11.3.2 Годинниковий алгоритм

Годинниковий алгоритм – це модифікація попереднього алгоритму (FIFO), який є занадто неефективним, оскільки постійно пересуває сторінки за списком. Тому краще зберігати всі сторінкові блоки в кільцевому списку у формі годинника, при цьому стрілка годинника вказує на найстарішу сторінку. У найпростішій схемі годинникової стратегії з кожним кадром зв'язується один додатковий біт, відомий як біт використання (u-use, або біт звернення – r-referenced).

Коли сторінка вперше завантажується в кадр, біт використання встановлюється рівним 1. При наступних зверненнях до сторінки, які викликали переривання через її відсутність, цей біт встановлюється рівним 1. При роботі алгоритму заміщення певна кількість кадрів, які є кандидатами на заміщення (поточний процес, локальна область видимості, вся основна пам'ять або глобальний контекст), розглядається як циклічний буфер, з яким пов'язаний покажчик. При заміщенні сторінки покажчик переміщається до наступного кадру в буфері (рис. 11.4).

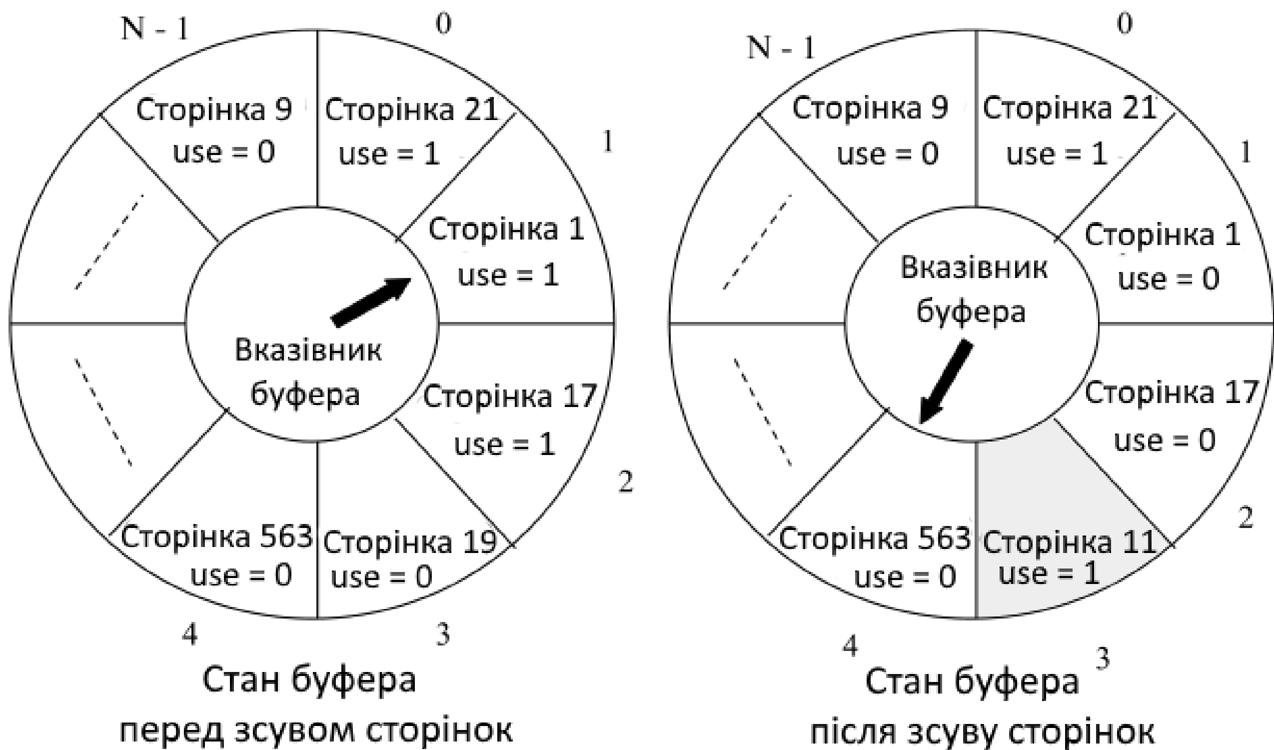


Рисунок 11.4 – Приклад работи годинникового алгоритму

Коли настає час заміщення сторінки, ОС сканує буфер для пошуку кадру, біт використання якого дорівнює 0. Всякий раз, коли в процесі пошуку зустрічається кадр з бітом використання, рівним 1, він скидається в 0. Перший же зустрінутий кадр з нульовим бітом використання вибирається для заміщення. Якщо усі кадри мають біт використання, рівний 1, покажчик здійснює повний круг і повертається до початкового положення, замінюючи сторінку в цьому кадрі. Як бачимо, ця стратегія схожа із стратегією «першим увійшов – першим вийшов», але відрізняється тим, що кадри, які мають встановлений біт використання, пропускаються алгоритмом. Буфер кадрів сторінок представлений у вигляді круга, звідки і пішла назва стратегії (інша назва – ***стратегія кругової заміни сторінок***). Ряд операційних систем використовує різні варіанти годинникової стратегії (наприклад, Multics).

На рис. 11.4 наведений простий приклад використання годинникової стратегії. Для заміщення доступні *N-1* кадр основної пам'яті, представлені у вигляді циклічного буфера. Безпосередньо перед тим, як замістити сторінку у буфері завантажуваною з вторинної пам'яті сторінкою 11, покажчик буфера вказує на кадр 1, що містить сторінку 1. Тепер приступимо до виконання годинникового алгоритму.

Оскільки біт використання сторінки 17 в кадрі 2 рівний 1, ця сторінка не заміщається. Замість цього її біт використання скидається, а покажчик переміщається до наступного кадру 3. Тут знаходиться сторінка 19, біт використання якої дорівнює 0. Ця сторінка вибирається для заміщення. На її місце завантажується сторінка 11, біт використання якої переводиться в 1. Покажчик перекладається на наступний кадр 4. На цьому виконання алгоритму завершується. На рис. 11.4 біт використання позначений як ***use***.

На рис. 11.5 наведено приклад роботи годинникового алгоритму, для якого використовується фіксований розподіл кадрів (три) і та ж черга сторінок, як і для попередніх алгоритмів (* – ***use=1***).

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Годинниковий алгоритм	2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
		3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
				1*	1	1	4*	4*	4	4	5*	5*
5 page faults					**F**	**F**	**F**		**F**		**F**	

Рисунок 11.5 – Приклад роботи годинникового алгоритму для трьох кадрів

Підвищити ефективність годинникового алгоритму можна шляхом збільшення кількості використовуваних при його роботі бітів. Такий алгоритм називають ***модифікованим годинниковим алгоритмом*** або ***алгоритмом не використовуваної останнім часом сторінки***.

У всіх процесорах, які підтримують сторінкову організацію, з кожною сторінкою в основний пам'яті (а отже, з кожним кадром) пов'язаний біт модифікації (***m***). Цей біт використовується для вказівки того, що дана сторінка не може бути заміщена до тих пір, поки її вміст не буде записано назад у вторинну пам'ять. Цей біт може використовуватися годинниковим алгоритмом наступним чином. Беручи до уваги біти використання і модифікації, всі кадри можна розділити на чотири категорії:

- використаний давно, не модифікований (u = 0, m = 0);
- використаний недавно, не модифікований (u = 1, m = 0);
- використаний давно, модифікований (u = 0, m = 1);
- використаний недавно, модифікований (u = 1, m = 1).

Використовуючи цю класифікацію, змінимо часовий алгоритм, який тепер буде описаний таким чином (рис. 11.6).

1. Скануємо буфер кадрів, починаючи з поточного положення. У процесі сканування біт використання не змінюється. Перша ж сторінка зі станом (*u = 0, m = 0*) заміщується.
2. Якщо виконання першого кроку алгоритму не увінчалося успіхом, шукаємо сторінку з параметрами (*u = 0, m = 1*). Якщо така сторінка знайдена, вона заміщується. У процесі виконання даного кроку у всіх переглянутих сторінок скидається біт використання.
3. Якщо виконання попереднього кроку не дало результату, покажчик повертається в початкове положення, але у всіх сторінок значення біта використання скинуто в 0. Повторимо крок 1 і, при необхідності, крок 2. Очевидно, на цей раз потрібна сторінка буде знайдена.

Такий алгоритм використаний в схемі віртуальної пам'яті ОС Macintosh. Позитивний момент цього алгоритму полягає, на відміну від простого годинникового алгоритму, в перевазі заміни сторінок, що не змінювалися, в порівнянні з заміною модифікованих сторінок.

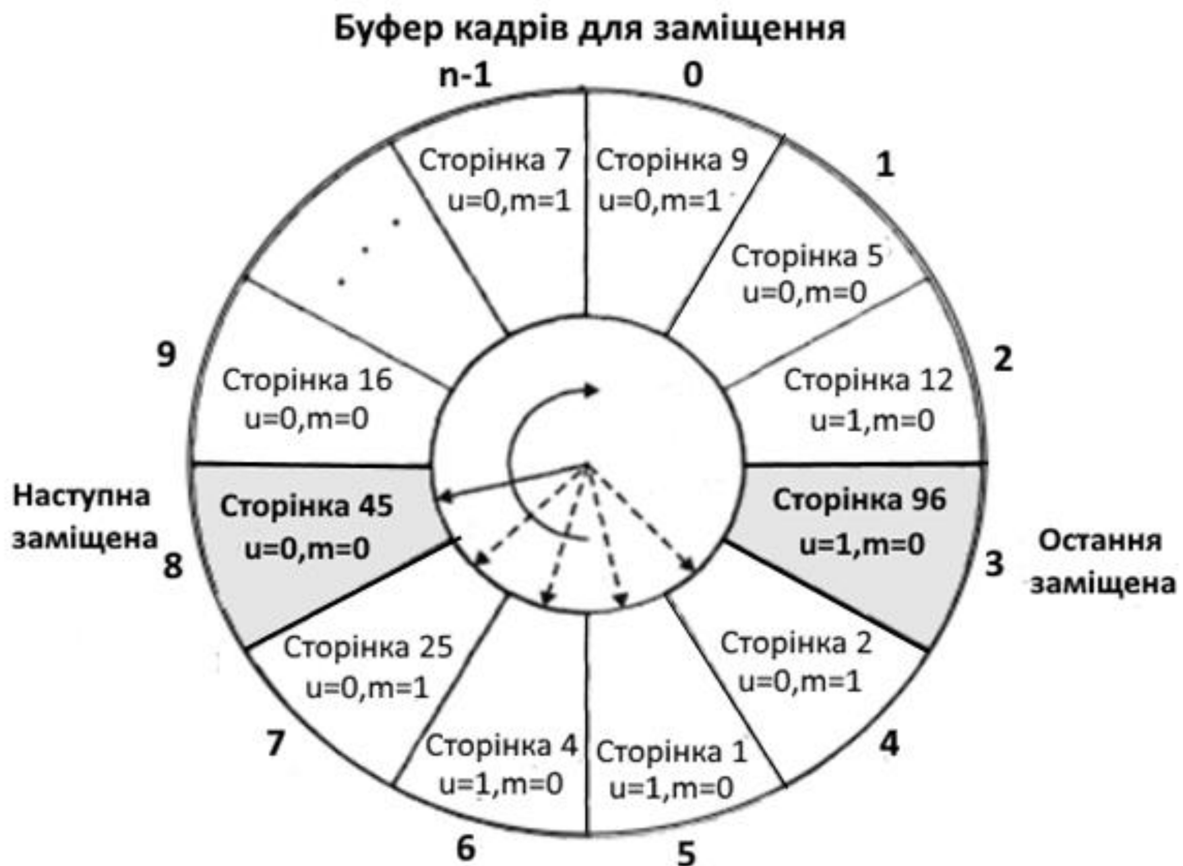


Рисунок 11.6 – Модифікований годинниковий алгоритм заміщення сторінок

Отже, годинниковий алгоритм циклічно проходить по всіх сторінках буфера в пошуках сторінки, яка не була модифікована з часу завантаження і давно не використовувалася. Така сторінка – хороший кандидат на заміщення, особливо з урахуванням того, що її не треба записувати на диск. Якщо при першому проході кандидатів на заміщення не знайшлося, алгоритм знову перевіряє буфер, тепер уже в пошуках модифікованої, давно не використовуваної сторінки. Хоча така сторінка і повинна бути записана перед заміщенням, відповідно до принципу локалізації вона навряд чи стане в нагоді в найближчому майбутньому. Якщо і цей прохід виявиться невдалим, всі сторінки позначаються як давно не використані, і виконується третій прохід.

За продуктивністю годинниковий алгоритм найближчий до алгоритму найдовше невикористовуваної сторінки (LRU).

11.3.3 Заміщення сторінки, яка найдовше не використовувлася

Одним з наближень до алгоритму OPT є алгоритм, що виходить з евристичного правила, що недавнє минуле – хороший орієнтир для прогнозування найближчого майбутнього. Наприклад, має сенс заміщати сторінку, яка не використовувалася впродовж найдовшого часу. Такий підхід називається ***Least Recently Used алгоритм (LRU)*** – сторінка, що не використалася найдовше. Згідно з принципом локалізації можна чекати, що ця сторінка не використовуватиметься і в найближчому майбутньому. Ця стратегія недалека від оптимальної. Робота алгоритму проілюстрована на рис. 11.7.

Порівнюючи цей алгоритм з іншими (з тим же потоком даних), можна побачити, що використання LRU алгоритму дозволяє скоротити кількість сторінкових порушень.

LRU – хороший, але важкий в реалізації алгоритм. Необхідно мати зв'язаний список усіх сторінок в пам'яті, на початку якого будуть зберігаться нещодавно використані сторінки. Причому цей список повинен оновлюватися при кожному зверненні до пам'яті.

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Алгоритм LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
4 page faults					**F**		**F**		**F**	**F**		

Рисунок 11.7 – Приклад роботи LRU алгоритму для трьох кадрів

Оскільки більшість сучасних процесорів не надають відповідної апаратної підтримки для реалізації алгоритму LRU, хотілося б мати алгоритм, досить близький до LRU, але такий, що не вимагає спеціальної підтримки.

Один з різновидів схеми LRU називається ***алгоритмом нечастого затребування*** – NFU (Not Frequently Used). Для цього алгоритму потрібний програмний лічильник, пов'язаний з кожною сторінкою в пам'яті, спочатку він рівний нулю. Під час кожного переривання по таймеру (а не після кожної інструкції) операційна система досліджує усі сторінки в пам'яті. Біт ***use*** кожної сторінки (він дорівнює 0 або 1) додається до лічильника, а потім прапор звернення скидається.

За допомогою лічильника намагаються відстежити, як часто відбувалося звернення до кожної сторінки. При сторінковому перериванні для заміщення вибирається сторінка з найменшим значенням лічильника.

Основна проблема, що виникає при роботі з алгоритмом NFU, полягає в тому, що він ніколи нічого не забуває. Наприклад, сторінка, до якої дуже часто зверталися впродовж деякого часу, а потім звертатися перестали, все одно не буде видалена з пам'яті, тому що її лічильник містить велику величину. Наприклад, у багатопрохідному компіляторі сторінки, які часто використовувалися під час першого проходу, можуть все ще мати високе значення лічильника при пізніших проходах. На щастя, невеликі зміни в алгоритмі дозволяють розв'язати цю проблему, яка дозволяє йому «забувати», досить добре моделювати алгоритм LRU:

– кожен лічильник зрушується вправо на один розряд перед збільшенням біта ***use***;
– біт ***use*** додається в крайній ліворуч, а не в крайній справа біт лічильника.

Коли відбувається сторінкове переривання, видаляється та сторінка, чий лічильник має найменшу величину. Ясно, що лічильник сторінки, до якої не було звернень. наприклад, за чотири тіка (tick – період таймера), розпочинатиметься з

чотирьох нулів і, таким чином, матиме нижче значення, ніж лічильник сторінки, на яку не посилалися впродовж тільки трьох тіків годинника.

На рис. 11.8 показано, як працює модифікований алгоритм, відомий ще як ***алгоритм старіння***. Припустимо, що після першого переривання від таймера біт **use(U)** для сторінок від 0 до 5 має, відповідно, значення 1, 0, 1, 0, 1 і 1. Іншими словами, між перериваннями від таймера, що відповідають тактам 0 і 1, було звернення до сторінок 0, 2, 4 і 5, в результаті якого їх біти U були встановлені в 1, а у інших сторінок їх значення залишилося рівним 0. Після того, як були зміщені значення шести відповідних лічильників і ліворуч було вставлено значення біта U, вони набули значень, показаних на рис. 11.8, *а.* У чотирьох стовпцях, що залишилися, показані стани шести лічильників після наступних чотирьох переривань від таймера.

Цей алгоритм відрізняється від алгоритму LRU двома особливостями. Розглянемо сторінки 3 і 5 на рис. 11.8, *д.* Ні до однієї з них за два переривання від таймера не було жодного звернення, але до обох було звернення за переривання від таймера, що передувало цим двом. Відповідно до алгоритму LRU якщо сторінка має бути видалена, то алгоритм повинен вибрати одну з цих двох сторінок. Проблема в тому, що не відомо, до якої з них зверталися в останню чергу між тактом 1 і тактом 2.

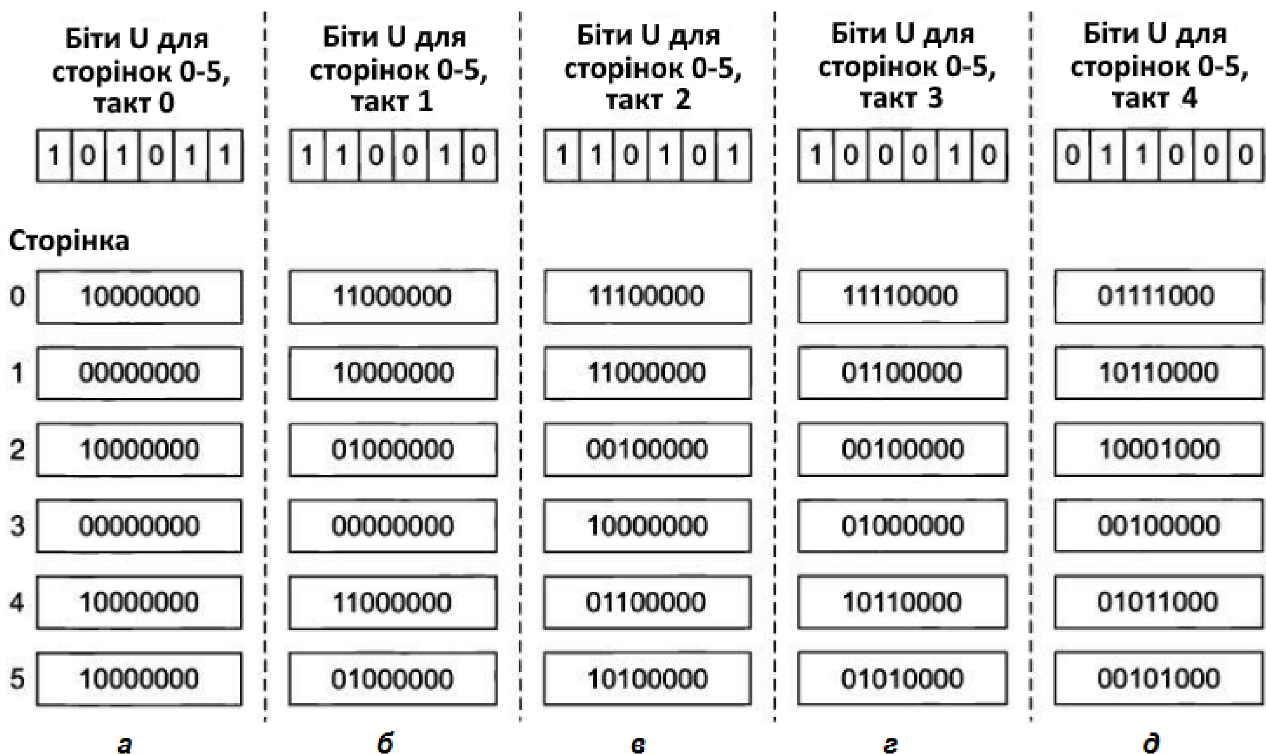


Рисунок 11.8 – Приклад роботи алгоритму старіння (NFU)

При записі тільки одного біта за інтервал між двома перериваннями від таймера ми втратили можливість відрізнити раніше звернення від пізнішого. Все, що ми можемо зробити, – це видалити сторінку 3, оскільки до сторінки 5 також було звернення двома тактами раніше, а до сторінки 3 такі звернення не були.

Друга відмінність між алгоритмом LRU і алгоритмом старіння полягає в тому, що в алгоритмі старіння лічильник має обмежену кількість біт (у цьому прикладі – 8 біт), яка звужує горизонт минулого, що переглядається ним. Припустимо, що у кожної з двох сторінок значення лічильника дорівнює нулю. Все, що ми можемо зробити, це вибрати одну з них довільним чином. Насправді цілком може виявитися, що до однієї з цих сторінок останнє звернення було 9 тактів назад, а до другої – 100 тактів назад. І цю обставину встановити неможливо. Але на практиці 8 біт цілком достатньо, якщо між перериваннями від таймера проходить приблизно 20 мс. Якщо до сторінки не було звернень впродовж 160 мс, то вона, напевно, вже не так важлива.

Ми розглянули декілька різних алгоритмів заміщення сторінок. Оптимальний алгоритм замінює ту сторінку, звернення до якої робилося раніше інших, що знаходяться в даний момент в пам'яті. На жаль, не існує способу визначення того, яка сторінка буде останньою, тому цей алгоритм не може використовуватися на практиці. Але він корисний в якості тестової задачі, відносно якого можна оцінювати інші алгоритми. На рис 11.9 зображена поведінка різних стратегій заміщення сторінок відносно оптимального алгоритму.

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Оптимальний алгоритм	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
3 page faults					**F**		**F**			**F**		
Алгоритм LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
4 page faults					**F**		**F**		**F**	**F**		
Алгоритм FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
6 page faults					**F**	**F**	**F**		**F**		**F**	**F**
Годинниковий алгоритм	2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
		3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
				1*	1	1	4*	4*	4	4	5*	5*
5 page faults					**F**	**F**	**F**		**F**		**F**	

Рисунок 11.9 – Поведінка чотирьох алгоритмів заміщення сторінок

11.3.4 Буферизація сторінок

Хоча алгоритми «найдовше невикористаний» і «годинниковий» перевершують алгоритм «першим увійшов – першим вийшов», вони обидва складні і мають високі накладні витрати в порівнянні з останнім. Крім того, слід враховувати, що вартість заміщення модифікованої сторінки перевищує

номінальну вартість заміщення немодифікованої сторінки, яку не треба записувати у вторинну пам'ять.

Є ще одна цікава стратегія, яка може підвищити продуктивність сторінкової організації при використанні найпростішого алгоритму заміщення. Це – буферизація сторінок, використана в VAX VMS. В якості алгоритму заміщення сторінок використовується найпростіший алгоритм «першим увійшов – першим вийшов». Для підвищення його продуктивності сторінка, що зміщується, не втрачається, а вноситься в один з двох списків: в список вільних сторінок, якщо сторінка не модифікувалася, або в список модифікованих сторінок. Зауважимо, що фізично сторінка не переміщується – замість цього її запис видаляється з таблиці сторінок і переноситься в список вільних або модифікованих сторінок.

Список вільних сторінок являє собою список кадрів сторінок, доступних для читання. VMS намагається постійно підтримувати деяку невелику кількість вільних кадрів. Коли сторінка зчитується в кадр, використовується кадр, розташований на початку списку; при цьому сторінка, яка перебувала в ньому раніше, знищується. При заміщенні немодифікованої сторінки вона залишається в пам'яті, а її кадр додається до кінця списку вільних сторінок; аналогічно, модифікована сторінка додається до списку модифікованих сторінок.

Важливим аспектом цих переміщень є те, що заміщувані сторінки залишаються в пам'яті. Отже, якщо процес звертається до такої сторінки, вона повертається в резидентну множину процесу без значних витрат. Насправді, списки вільних і модифікованих сторінок працюють в якості кеша сторінок. Список модифікованих сторінок дозволяє записувати їх не по одній, а кластерами, що істотно знижує кількість операцій введення-виведення, а, отже, і час звернення до диска.

11.3.5 Стратегія заміщення і розмір кеша

Як зазначалося раніше, розмір основної пам'яті з часом стає все більше, як і розмір додатків. Втіхою може служити те, що розміри кешів також збільшуються. При використанні кешів великого розміру заміщення сторінок віртуальної пам'яті може впливати на продуктивність. Якщо кадр сторінки, що обраний для заміщення, розташовується в кеші, то разом з втратою сторінки з блоку кеша втрачається весь блок.

У системах з використанням буферизації того чи іншого виду продуктивність кеша можна збільшити шляхом додавання до стратегії заміщення стратегію розміщення сторінок у буфері. Більшість операційних систем розміщують сторінки в буфері в довільних кадрах, як правило, з використанням алгоритму «першим увійшов – першим вийшов». Дослідження показали, що правильний вибір стратегії розміщення може призвести до зменшення неуспішних пошуків в кеші на 10-20%.

Суть цих стратегій полягає в розміщенні послідовних сторінок в основній пам'яті таким чином, щоб мінімізувати кількість кадрів сторінок, що відображаються в одні і ті ж слоти кеша.

Контрольні питання і тести до розділу 11

Контрольні питання

1. З реалізацією яких стратегій пов'язано програмне забезпечення підсистеми управління пам'яттю ОС?
2. Якими одиницями оцінюється ефективність алгоритмів заміщення сторінок?
3. Оптимальний алгоритм призводить до мінімальної кількості переривань через відсутність сторінки. Чому неможливо реалізувати такий алгоритм?
4. У якому алгоритмі заміщення сторінок певні послідовності звернень до сторінок призводять до «аномалії Беледі»?
5. На чому базується стратегія кругової заміни сторінок?
6. За рахунок чого можна підвищити ефективність годинникового алгоритму?
7. З яких основних кроків складається модифікований годинниковий алгоритм?
8. На якому принципі локалізації розроблений алгоритм заміщення сторінок LRU (сторінка, яка не використовувалася найдовше)?
9. Один з різновидів схеми LRU називається алгоритмом нечастого затребування – NFU (Not Frequently Used). На чому базується його стратегія?

Тести

1. При сторінковому перериванні і відсутності вільних блоків фізичної пам'яті операційна система повинна вибрати:
   1) сторінкуу-кандидат на видалення з пам'яті і зберегти сторінку, що видаляється, на диску;
   2) сторінку, яка не змінювалася, і зберегти сторінку, що видаляється, на диску;
   3) сторінку-кандидат на видалення з пам'яті і зберегти копію сторінки, що видаляється, в таблиці сторінок;
   4) сторінку-кандидат на видалення з пам'яті і зберегти сторінку, що видаляється, на диску, якщо вона зазнала зміни.
2. Повна реалізація алгоритму LRU (Least Recently Used):
   1) практично скрутна;
   2) теоретично неможлива;
   3) можлива за умови побудови таблиці сторінок у вигляді бінарних дерев;
   4) можлива при використанні стекової організації таблиці сторінок.
3. Для деякого процесу відомий такий рядок запитів сторінок пам'яті 4, 3, 4, 1, 5, 4, 2, 5, 3, 4, 5, 4. Скільки ситуацій відмови сторінки (page fault) виникне для цього процесу при використанні алгоритму заміщення сторінок OPT (оптимальний алгоритм) і трьох сторінкових кадрів?
   1) 4;
   2) 3;
   3) 2;
   4) 5.

4. Для деякого процесу відомий такий рядок запитів сторінок пам'яті 4, 3, 4, 1, 5, 4, 2, 5, 3, 4, 5, 4. Скільки ситуацій відмови сторінки (page fault) виникне для цього процесу при використанні алгоритму заміщення сторінок FIFO (First Input First Output) і трьох сторінкових кадрів?
   1) 4;
   2) 5;
   3) 6;
   4) 7.
5. Для деякого процесу відомий наступний рядок запитів сторінок пам'яті 4, 3, 4, 1, 5, 4, 2, 5, 3, 4, 5, 4. Скільки ситуацій відмови сторінки (page fault) виникне для цього процесу при використанні алгоритму заміщення сторінок Least Recently Used (LRU) і трьох сторінкових кадрів?
   1) 5;
   2) 6;
   3) 4;
   4) 3.
6. Для деякого процесу відомий наступний рядок запитів сторінок пам'яті 4, 3, 4, 1, 5, 4, 2, 5, 3, 4, 5, 4. Скільки ситуацій відмови сторінки (page fault) виникне для цього процесу при використанні годинникового алгоритму заміщення сторінок і трьох сторінкових кадрів?
   1) 4;
   2) 5;
   3) 6;
   4) 7.
7. Який з алгоритмів заміщення сторінок реалізується заміщенням першої сторінки, що прийшла?
   1) LRU;
   2) FIFO;
   3) OPT;
   4) NFU.
8. Який з алгоритмів заміщення сторінок реалізується заміщенням сторінки, не використаної впродовж найбільшого періоду часу?
   1) LRU;
   2) FIFO;
   3) OPT;
   4) NFU.
9. Який з алгоритмів заміщення сторінок неможливо реалізувати на практиці?
   1) LRU;
   2) FIFO;
   3) OPT;
   4) NFU.

12 ПЛАНУВАННЯ ПРОЦЕСІВ

За часів пакетної обробки, алгоритм планування був простий: запустити таку задачу на стрічці або з перфокарт. З появою систем розподілу часу алгоритм планування ускладнився, оскільки тепер декілька задач очікували обслуговування. На деяких мейнфреймах досі поєднуються системи пакетної обробки, і служби розподілу часу. У результаті планувальник повинен вирішувати запускати наступне пакетне завдання чи надати процесор інтерактивному завданню.

З появою ПК ситуація змінилася. По-перше, більшу частину часу активний тільки один процес. По-друге, ПК стали настільки швидше, що час процесора вже став не таким дефіцитним ресурсом. Більшість програм обмежені швидкістю, з якою користувач вводить вхідні дані (з клавіатури або за допомогою миші), а не швидкістю процесора. Навіть процедури компіляції, основні споживачі процесорного часу, тепер займають кілька секунд.

Картина також змінилася, коли появились потужні робочі станції і сервери. Тут планування знову грає істотну роль, оскільки кілька процесів намагаються отримати доступ до ресурсу. Наприклад, коли процесору потрібно вибрати між процесом перемальовування екрану після того, як користувач закрив вікно додатку, і процесом, що відсилає пошту, враження користувача від реакції комп'ютера буде істотно залежати від цього вибору. Адже, якщо перемальовування екрану під час відправки пошти займе 2 секунди, то користувач вирішить, що система дуже повільна, тоді як 2-х секундне відсилання пошти навіть не помітять. У цьому випадку планування процесів дуже важливе.

Можливість паралельного виконання процесів (потоків) залежить від кількості доступних процесорів. Якщо процесор один, паралельне виконання неможливе (в кожен момент часу може виконуватися тільки один процес). Якщо процесорів $N > 1$, паралельне виконання може бути реалізовано для N процесів (потоків, по одному на процесор).

12.1 Планування в системах з одним процесором

У багатозадачних системах в основний пам'яті одночасно міститься код декількох процесів. У роботі кожного процесу періоди використання процесора чергуються з очікуванням завершення виконання операцій введення-виведення або деяких зовнішніх подій. Процесор зайнятий виконанням одного процесу, в той час як інші перебувають в стані очікування.

ОС повинна вирішувати задачу планування (scheduling), основна мета якої для однопроцесорної системи полягає в такій організації виконання процесів, через яку у користувача системи виникає враження, що вони виконуються одночасно.

Для організації управління процесами необхідно врахувати щонайменше два основних аспекти: визначення рівня, на якому виконується планування процесів і вибір алгоритму планування.

Таким чином, ключем до багатозадачності є планування. Як правило, використовуються три види (рівні) планування:

1. Довгострокове планування (планування верхнього рівня).
2. Середньострокове планування (планування проміжного рівня).
3. Короткострокове планування (планування нижнього рівня).

12.1.1 Рівні планування процесів

Однією з важливих задач, яку вирішує ОС, є проблема, пов'язана з визначенням коли і яким процесам слід виділяти ресурси процесора – задача планування завантаження процесора. Планування впливає на продуктивність системи, оскільки саме воно визначає, який процес буде працювати, а який повинен буде очікувати на виконання. Існують три рівні такого планування (рис. 12.1).

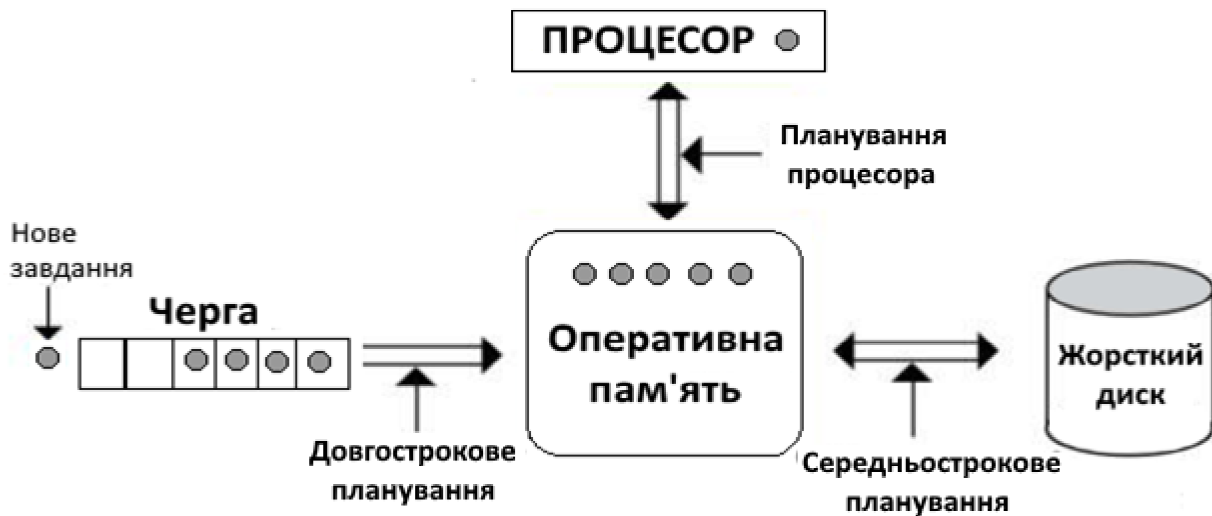


Рисунок 12.1 – Рівні планування

Довгострокове планування. Довгострокове планування (або планування завдань) вказує, які програми (завдання) допускаються до виконання системою і тим самим визначає ступінь багатозадачності. Будучи допущеним до виконання, завдання стає процесом, який додається в чергу для короткострокового планування. У деяких системах новостворений процес додається до черги середньострокового планування, будучи цілком скинутим на диск.

Планування завдань з'явилося в пакетних системах після того, як для зберігання сформованих пакетів завдань почали використовуватися магнітні диски. Магнітні диски, будучи пристроями прямого доступу, дозволяють завантажувати завдання в комп'ютер у довільному порядку, а не тільки в тому, в якому вони були записані на диск. Змінюючи порядок завантаження завдань в обчислювальну систему, можна підвищити ефективність її використання.

Рішення про те, коли треба створити новий процес, визначаються загальним рівнем багатозадачності. Чим більше процесів буде створено, тим менший відсоток часу витрачатиметься на виконання кожного з них, але повніше буде завантажений процесор. Кожен раз при завершенні завдання довгостроковий планувальник вирішує, чи треба додавати в систему один або кілька нових процесів. Крім того, довгостроковий планувальник може бути

викликаний в разі, коли відносний час простою процесора перевищує певний поріг.

Рішення про те, яке з завдань має бути додано в систему, може грунтуватися на простому принципі «першим прийшов – першим обслуговується». Крім того, для управління продуктивністю системи може використовуватися і спеціальний інструментарій, який враховує пріоритет задач, час очікуваного виконання і вимоги для роботи пристроїв введення-виведення.

У разі використання інтерактивних програм в системах з розподіленням часу запит на запуск процесу може здійснюватися внаслідок обслуговування підключення до системи. ОС приймає всіх зареєстрованих користувачів до насичення системи (поріг визначається заздалегідь). Після досягнення системою стану насичення на всі запити на вхід в систему, буде отримано повідомлення про заповнення системи і тимчасово припинення доступу до неї з пропозицією повторити операцію входу пізніше.

Якщо ступінь мультипрограмувания системи підтримується постійною, тобто середня кількість процесів укомп'ютері не змінюється, то нові процеси можуть з'являтися тільки після завершення раніше завантажених. Тому довгострокове планування здійснюється досить нечасто, між появою нових процесів можуть проходити хвилини і навіть десятки хвилин. Звідси і назва цього рівня планування – ***довгострокове***.

Середньострокове планування. У деяких обчислювальних системах буває вигідно для підвищення їх продуктивності тимчасово видалити будь-який процес, який виконуєтся, з оперативної пам'яті на диск, а пізніше повернути його назад для подальшого виконання. Така процедура в англомовній літературі отримала назву *swapping*, що можна перевести як перекачування (підкачка), хоча у фаховій літературі воно вживається без перекладу – ***свопінг***. Коли і який із процесів потрібно перекачати на диск і повернути назад, вирішується додатковим проміжним рівнем планування процесів – ***середньостроковим***.

Рішення про завантаження процесу в пам'ять приймається в залежності від ступеня багатозадачності. Крім того, в системі з відсутністю віртуальної пам'яті середньострокове планування також тісно пов'язане з питаннями управління пам'яттю. Таким чином, рішення про завантаження процесу в пам'ять має враховувати вимоги до пам'яті вивантажуваного процесу.

Середньострокове планування керує переходом процесів з призупинених станів в стан готовності і назад. Керуючі блоки готових до виконання процесів організовуються в пам'яті в структуру, яку називають чергою готових процесів. Детальніше розглянемо ці черги під час короткострокового планування.

Короткострокове планування. Короткострокове планування (**диспетчеризація** або **планування процесора**) є найважливішим видом планування. Розглядаючи частоту роботи планувальника, можна сказати, що довгострокове планування виконується відносно нечасто, середньострокове – дещо частіше. Короткостроковий планувальник, відомий також як **диспетчер**, працює найчастіше, визначаючи, який саме процес буде виконуватися наступним. Вибір нового процесу для виконання впливає на функціонування системи до настання чергової аналогічної події.

Короткостроковий планувальник викликається при настанні події, що може призупинити поточний процес або надати можливість припинити виконання даного процесу на користь іншого. Приклади таких подій: переривання таймера, переривання введення-виведення, виклики ОС, сигнали.

Всі стратегії і алгоритми планування, які ми будемо розглядати далі, належать до короткострокового планування (диспетчеризації).

Місце планування в графі станів і переходів процесів показано на рис. 12.2.

Рисунок 12.2 – Місце планування в графі процесів

12.1.2 Параметри планування

Для здійснення поставлених цілей розумні алгоритми планування повинні спиратися на будь-які характеристики процесів у системі, завдань в черзі на завантаження, стану самої обчислювальної системи, іншими словами, на параметри планування. У цьому розділі ми опишемо ряд таких параметрів, не претендуючи на повноту викладу.

Усі параметри планування можна розбити на дві великі групи: статичні і динамічні. Статичні параметри не змінюються в ході функціонування обчислювальної системи, динамічні ж, навпаки, схильні до постійних змін.

До статичних параметрів обчислювальної системи можна віднести граничні значення її ресурсів вже на етапі завантаження (розмір оперативної пам'яті, максимальна кількість пам'яті на диску для здійснення свопінгу, кількість підключених пристроїв введення-виведення тощо).

1. Яким користувачем запущений процес або сформовано завдання.
2. Наскільки важливою є поставлена задача, тобто, її пріоритет.
3. Скільки процесорного часу запрошено користувачем для задачі.

4. Яке співвідношення процесорного часу і часу, необхідного для здійснення операцій введення-виведення.
5. Які ресурси обчислювальної системи (оперативна пам'ять, пристрої введення-виведення, спеціальні бібліотеки, системні програми тощо) і в якій кількості необхідні завданню.

Динамічні параметри системи описують кількість вільних ресурсів у поточний момент часу.

Алгоритми довгострокового планування використовують у своїй роботі статичні і динамічні параметри обчислювальної системи і статичні параметри процесів (динамічні параметри процесів на етапі завантаження завдань ще не відомі). Алгоритми короткострокового і середньострокового планування додатково враховують і динамічні характеристики процесів. Для середньострокового планування в якості таких характеристик, наприклад, може виступати інформація.

1. Скільки часу пройшло з моменту вивантаження процесу на диск або його завантаження в оперативну пам'ять.
2. Скільки оперативної пам'яті займає процес.
3. Скільки процесорного часу вже було надано процесу.

Для короткострокового планування нам знадобиться ввести ще два динамічних параметри. Діяльність будь-якого процесу можна представити як послідовність циклів використання процесора і очікування завершення операцій введення-виведення. Проміжок часу безперервного використання процесора носить англійською мовою назву ***CPU burst***, а проміжок часу безперервного очікування введення-виведення – ***I/O burst***. На рис. 12.2 показаний фрагмент діяльності деякого процесу на псевдомові програмування з виділенням вказаних проміжків. Для стислості викладу ми будемо використовувати терміни ***CPU burst*** і ***I/O burst*** без перекладу. Значення тривалості останніх і чергових *CPU burst* і *I/O burst* є важливими динамічними параметрами процесу.

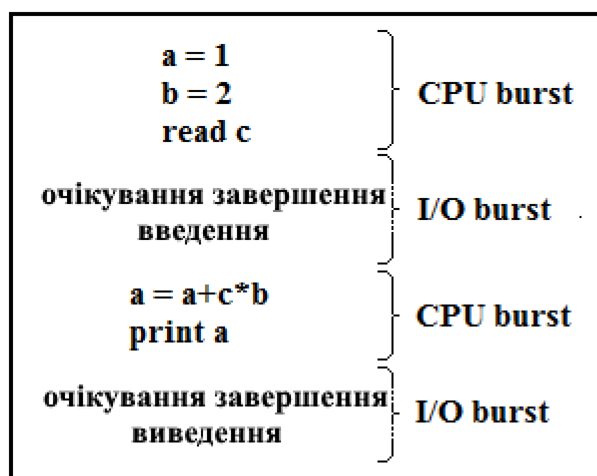


Рисунок 12.2 – Фрагмент діяльності процесу з виділенням проміжків безперервного використання процесора і очікування введення-виведення

12.1.3 Алгоритми планування

Основна мета короткострокового планування полягає в розподілі процесорного часу таким чином, щоб оптимізувати один або кілька аспектів поведінки системи. Існує певна кількість критеріїв оцінки різних стратегій планування. Найпоширеніші критерії можуть бути класифіковані в двох площинах.

По-перше, розділимо їх на **призначені для користувача** і **системні**. Призначені для користувача критерії пов'язані з поведінкою системи стосовно окремого користувача або процесу. Як приклад можна навести час відгуку в інтерактивній системі – це інтервал часу між передачею запиту і початком відповіді на нього. Його користувач відчуває безпосередньо. Для забезпечення якісного сервісу для користувачів треба встановити поріг відгуку, наприклад, 2 с. Тоді мета механізму планування повинна складатися в максимізації кількості користувачів, середній час відгуку для яких не перевищує 2 с.

Системні критерії орієнтовані на ефективність і повноту використання процесора (***пропускна здатність***). Як приклад можна навести пропускну здатність, яка представляє собою швидкість завершення процесів. Це, безумовно, ефективна міра продуктивності системи, яка повинна бути максимальною. Однак вона більшою мірою орієнтована на продуктивність системи, а не на обслуговування користувача. Критерії, які орієнтовані на продуктивність, можуть бути виражені чисельними значеннями, і легко вимірюватися.

З іншого боку, призначені для користувача критерії важливі майже для всіх систем, системні критерії для систем одного користувача не так значимі. В цьому випадку, мабуть, досягнення високої ефективності використання процесора або висока продуктивність не так істотні, як швидкість відповіді системи додатку користувача. Назвемо ключові критерії планування.

Критерії, які призначені для користувача і пов'язані з продуктивністю:

1. ***Час обороту*** – інтервал часу між подачею процесу і його завершенням. Включає час виконання, а також час на очікування ресурсів, у тому числі і процесора (для пакетних задач).
2. ***Час відгуку*** – в інтерактивних процесах це час, який минув між поданням запиту і початком отримання відповіді на нього. Стратегія планування повинна спробувати скоротити час отримання відповіді при максимізації кількості інтерактивних користувачів, час відгуку для яких не виходить за задані межі.
3. ***Граничний термін*** – у разі зазначення граничного терміну завершення процесу планування має підпорядкувати йому всі інші цілі максимізації кількості процесів, що завершуються в указаний термін.

Критерії, які призначені для користувача, інші:

Передбачуваність **–** певна задача має виконуватися за один і той же час незалежно від завантаження системи. Великі зміни часу виконання або часу відгуку дезорієнтують користувачів. Це може сигналізувати про велику

завантаженість ОС або про необхідність додаткового налаштування системи для усунення нестабільності її роботи.

Системні критерії, пов'язані з продуктивністю:

1. ***Пропускна здатність*** – стратегія планування повинна намагатися максимізувати кількість процесів, що завершуються за одиницю часу. Це значення залежить від середньої тривалості процесу, але при цьому на нього впливає і використовувана стратегія планування.
2. ***Використання процесора*** **–** це відсоток часу, коли процесор зайнятий.

Системні, інші критерії:

1. ***Неупередженість*** – при відсутності додаткових вказівок від користувача або системи всі процеси повинні розглядатися як рівнозначні і жоден з них не повинен «голодувати».
2. ***Використання пріоритетів*** – якщо процесам призначені пріоритети, то стратегія планування повинна віддавати перевагу процесам з більш вищим пріоритетом.
3. ***Баланс ресурсів*** **–** стратегія планування повинна підтримувати зайнятість системних ресурсів. Перевагу треба надати процесу, який недостатньо використовує важливі ресурси. Цей пріоритет включає використання довгострокового і середньострокового планування.

Всі ці критерії планування взаємозалежні, і досягти оптимального результату по кожному з них одночасно неможливо. Наприклад, забезпечення хорошого відгуку може зажадати застосування алгоритму з високою частотою перемикання процесів, що підвищить накладні витрати і, отже, знизить пропускну здатність системи. Отже, розробка стратегії планування являє собою пошук компромісу серед суперечливих вимог.

Розрізняють дві основні стратегії планування – **витісняючу** і **невитісняючу багатозадачність**. При витісняючій багатозадачності процеси, які виконуються, можуть бути перервані планувальником ОС без їх участі для передачі управління іншому процесу. Найчастіше це здійснюється оброблювачем переривань від системного таймера. Така стратегія реалізована в усіх сучасних ОС.

При невитісняючій багатозадачності процес може виконуватися протягом необмеженого часу і не може бути перерваний ОС. Процеси самі віддають управління ОС, або переходити в стан очікування. Така стратегія була реалізована в MS Windows 3.1 і ОС Novell Net Ware 3.11 в 90-і роки.

Основною відмінністю між витісняючими і невитісняючими алгоритмами є ступінь централізації механізму планування процесів (потоків). При витісняючому мультипрограмуванні функції планування процесів цілком зосереджені в операційній системі. Програміст пише свій додаток, не піклуючись про те, що він буде виконуватися одночасно з іншими задачами. При цьому операційна система виконує такі функції: визначає момент зняття з виконання активного процесу, запам'ятовує його контекст, вибирає з черги готових процесів наступний, запускає новий процес на виконання, завантажуючи його контекст.

При невитісняючому мультипрограмуванні механізм планування розподілений між операційною системою і прикладними програмами.

Прикладна програма, отримавши управління від операційної системи, сама визначає момент завершення чергового циклу свого виконання і тільки потім передає керування ОС за допомогою будь-якого системного виклику. ОС формує черги процесів і вибирає відповідно до деякого правила (наприклад, з урахуванням пріоритетів) наступний процес на виконання. Такий механізм створює проблеми як для користувачів, так і для розробників додатків.

Для користувачів це означає, що управління системою втрачається на довільний період часу, який визначається додатком (а не користувачем). Якщо додаток витрачає занадто багато часу на виконання будь-якої роботи, наприклад на форматування диска, користувач не може переключитися з задачі на іншу задачу, наприклад, на текстовий редактор, в той час як форматування тривало б у фоновому режимі.

Тому розробники додатків для операційного середовища з витісняючою багатозадачністю змушені, покладаючи на себе частину функцій планувальника, створювати додатки так, щоб вони виконували свої задачі невеликими частинами. Наприклад, програма форматування може відформатувати одну доріжку дискети і повернути управління системі. Після виконання інших задач система поверне управління програмою форматування, щоб та відформатувати наступну доріжку.

Подібний метод поділу часу між задачами працює, але він істотно ускладнює розробку програм і висуває підвищені вимоги до кваліфікації програміста. Програміст повинен забезпечити «дружнє» ставлення своєї програми до інших виконуваних одночасно з нею програм. Для цього в програмі повинні бути передбачені часті передачі управління операційній системі. Крайнім виявом «не дружелюбності» додатка є його зависання, яке призводить до загального краху системи. У системах з витіснячою багатозадачністю такі ситуації, як правило, виключені, так як центральний планувальний механізм має можливість зняти зависле завдання з виконання.

Однак розподіл функцій планування потоків між системою і додатками не завжди є недоліком, а за певних умов може бути і перевагою, тому що дає можливість розробнику додатків самому проектувати алгоритм планування, найбільш підходящий для даного фіксованого набору завдань. Так як розробник сам визначає в програмі момент повернення управління, то при цьому виключаються нераціональні переривання програм в «незручні» для них моменти часу.

Крім того, легко вирішуються проблеми спільного використання даних: завдання під час кожного циклу виконання використовує їх монопольно і впевнено, що протягом цього періоду ніхто інший не змінить дані. Істотною перевагою невитісняючого планування є вища швидкість перемикання з процесу (потоку) на процес.

Майже в усіх сучасних ОС, орієнтованих на високу продуктивність виконання додатків (UNIX, Windows NT/2000/XP, OS/2), реалізовані витісняючі алгоритми планування потоків (процесів). Прикладом ефективного використання невитісняючого планування є файл-сервери NetWare 3.x/4.x, в яких досягнута висока швидкість виконання файлових операцій.

12.2 Алгоритми планування, засновані на квантуванні

В основі багатьох витісняючих алгоритмів планування лежить концепція квантування. Відповідно до цієї концепції кожному потоку по черзі для виконання надається обмежений безперервний період процесорного часу – квант.

Квант **–** це часовий інтервал, протягом якого процесу дозволено займати процесор, тобто дозволено перебувати в стані виконання. Поняття кванта грунтується на періоді таймера, який називається тіком. Тік (tick) – мінімально можливий часовий інтервал, який дорівнює дискретному проміжку часу між двома сигналами таймера, що генеруються через кожні 55мс.

Квант дорівнює цілому числу тіків у проміжку від 1 до 255. Збільшення кванта уповільнює реакцію системи при збільшенні черги, а зменшення – збільшує частку накладних часових витрат на переключення процесів. Для ефективної роботи системи необхідно знаходити розумний компроміс.

Відповідно до алгоритмів, заснованих на квантуванні, зміна активного потоку відбувається, якщо:

- потік завершився і залишив систему;
- виникла помилка;
- потік перейшов в стан очікування;
- вичерпаний квант процесорного часу, відведений даному потоку.

Потік, який вичерпав свій квант, переводиться в стан готовність і очікує, коли йому буде надано новий квант процесорного часу, а на виконання відповідно до визначеного правила вибирається новий потік з черги готових. Таким чином, жоден процес не займає процесор надовго, і з цієї причини квантування широко використовується в системах розподілу часу. Граф станів потоку, зображений на рис. 12.3, відповідає алгоритму планування, заснованому на квантуванні.

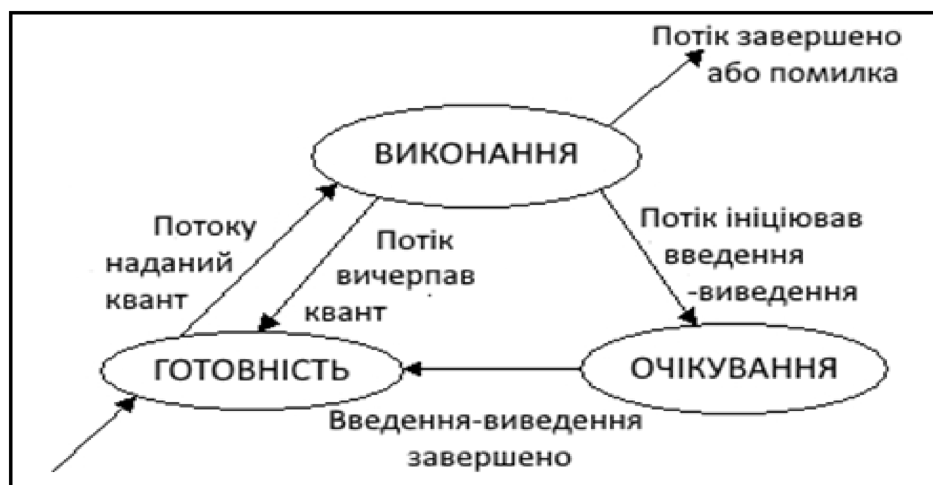


Рисунок 12.3 – Граф станів потоку в системі з квантуванням

Кванти, що виділяються потокам, можуть бути однаковими для всіх потоків або змінюватися в різні періоди життя процесу. Розглянемо випадок, коли всім потокам надаються кванти однакової довжини q (рис. 12.4).

Якщо в системі є *n* потоків, то час, який потік проводить в очікуванні наступного кванта, можна грубо оцінити як *q (nl)*. Чим більше потоків в системі, тим більше час очікування, тим менше можливості у кількох користувачів вести одночасну інтерактивну роботу.

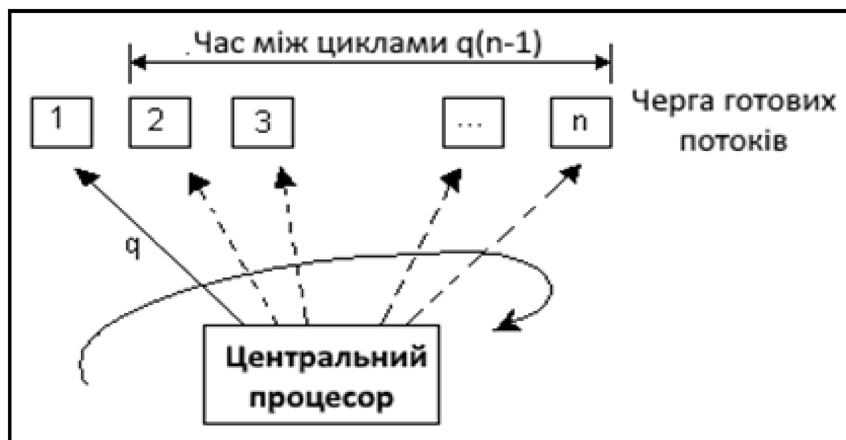


Рисунок 12.4 – Ілюстрація розрахунку часу очікування в черзі

Але якщо величина кванта обрана дуже невеликою, то значення *q (nl)* все одно буде досить мале для того, щоб користувач не відчував дискомфорту від присутності в системі інших користувачів. Типове значення кванта в системах поділу часу становить десятки мілісекунд.

Якщо квант короткий, то сумарний час, який проводить потік в очікуванні процесора, прямо пропорційній часу, необхідному для його виконання (тобто часу, який треба було б надати для виконання цього потоку при монопольному використанні обчислювальної системи).

Чим більше квант, тим вище ймовірність того, що потоки завершаться в результаті першого ж циклу виконання, і тим менше явною стає залежність часу очікування потоків від їх часу виконання. При досить великому кванті часу алгоритм квантування вироджується в алгоритм послідовної обробки, властивий однопрограмним системам, при якому час очікування завдання в черзі взагалі ніяк не залежить від його тривалості.

Потоки отримують для виконання квант часу, але деякі з них використовують його не в повному обсязі, наприклад через необхідність виконати введення або виведення даних. У результаті виникає ситуація, коли потоки з інтенсивними зверненнями до введення-виведення використовують тільки невелику частину виділеного їм процесорного часу. Алгоритм планування може виправити цю «несправедливість».

В якості компенсації за повністю невикористані кванти потоки отримують привілеї при подальшому обслуговуванні. Для цього планувальник створює дві черги готових потоків (рис. 12.5).

Черга 1 утворена потоками, які прийшли в стан готовності в результаті вичерпання кванта часу, а черга 2 – потоками, в яких завершилася операція введення-виведення. При виборі потоку для виконання насамперед проглядається друга черга, і тільки якщо вона порожня, квант виділяється потоку з першої черги.

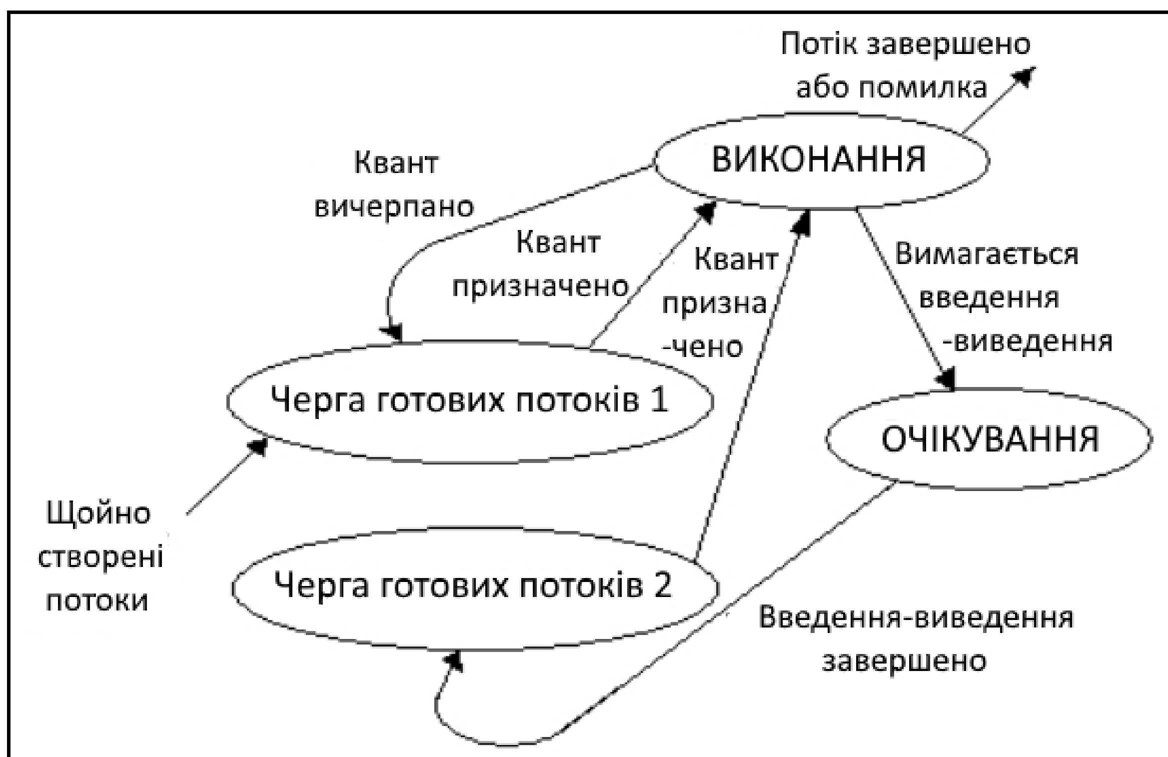


Рисунок 12.5 – Квантування з перевагою потоків, які інтенсивно звертаються до пристроїв введення-виведення

Багатозадачні ОС втрачають деяку кількість процесорного часу для виконання допоміжних робіт під час перемикання контекстів задач. При цьому запам'ятовуються і відновлюються регістри, прапори і покажчики стека. Витрати на ці допоміжні дії залежать від величини кванта часу, тому чим більше квант, тим менше сумарні накладні витрати, пов'язані з перемиканням потоків.

В алгоритмах, заснованих на квантуванні, яку б мету вони не переслідували (перевага коротких або довгих завдань, компенсація недовикористаного кванта), не використовується ніякої попередньої інформації про завдання. При надходженні завдання на обробку ОС не має ніяких відомостей про те, чи є воно коротким, чи довгим, наскільки інтенсивними будуть його запити до пристроїв введення-виведення тощо.

12.3 Алгоритми планування, засновані на пріоритетах

Іншою важливою концепцією, що лежить в основі багатьох витісняючих алгоритмів планування, є пріоритетне обслуговування. Пріоритетне обслуговування передбачає наявність у потоків деякої відомої характеристики – пріоритету, на підставі якої визначається порядок їх виконання.

Пріоритет – це число, що характеризує ступінь привілейованості потоку при використанні ресурсів обчислювальної машини, зокрема процесорного часу: чим вище пріоритет, тим вище привілеї, тим менше часу буде проводити потік в чергах.

Пріоритет може виражатися цілим або дробовим, додатнім або від'ємним числом. У деяких ОС прийнято, що пріоритет потоку тим вище, чим більше

число (в арифметичному сенсі), що позначає пріоритет. В інших системах, навпаки, чим менше число, тим вище пріоритет.

Так, в UNIX і багатьох інших системах великі значення пріоритетів відповідають процесам з низьким пріоритетом. Деякі системи, такі як OS/2 або Windows, використовують зворотнє значення: більше значення вказує на вищий пріоритет.

У більшості операційних систем, що підтримують потоки, пріоритет потоку безпосередньо пов'язаний з пріоритетом процесу, в рамках якого виконується даний потік. Пріоритет процесу назначається операційною системою при його створенні. Значення пріоритету включається в описувач процесу і використовується при призначенні пріоритету потокам цього процесу. При призначенні пріоритету новоствореному процесу ОС враховує, чи є цей процес системним чи прикладним, який статус користувача, що запустив процес, чи була явна вказівка користувача на присвоєння процесу певного рівня пріоритету. Потік може бути ініційований не тільки командою користувача, але і в результаті виконання системного виклику іншим потоком. У цьому випадку при призначенні пріоритету новому потоку ОС повинна брати до уваги значення параметрів системного виклику.

У багатьох ОС передбачається можливість зміни пріоритетів протягом життя потоку. Зміни пріоритетів можуть відбуватися з ініціативи самого потоку, коли він звертається з відповідним викликом до операційної системи. Пріоритет може призначатися директивно адміністратором системи в залежності від важливості, або обчислюватися самою ОС за певними правилами. Пріоритет може також змінюватися з ініціативи користувача, коли він виконує відповідну команду. Крім того, ОС сама може змінювати пріоритети потоків в залежності від ситуації, що складається в системі. В останньому випадку пріоритети називаються ***динамічними***.

Від того, які пріоритети призначені потокам, істотно залежить ефективність роботи всієї обчислювальної системи. У сучасних ОС, щоб уникнути розбалансування системи, яке може виникнути при неправильному призначенні пріоритетів, можливості користувачів впливати на пріоритети процесів і потоків намагаються обмежувати. Права підвищувати пріоритети потокам дозволено робити тільки адміністраторам.

Як приклад розглянемо схему призначення пріоритетів потокам, прийняту в операційній системі Windows NT (рис. 12.6). В системі визначено 32 рівні пріоритетів і два класи потоків – потоки реального часу і потоки зі змінними пріоритетами. Діапазон від 1 до 15 включно відведений для потоків зі змінними пріоритетами, а від 16 до 31 – для критичніших потоків реального часу (пріоритет 0 зарезервований для системних цілей).

При створенні процесу він у залежності від класу отримує за замовчуванням базовий пріоритет у верхній або нижній частині діапазону. Базовий пріоритет процесу в подальшому може бути підвищений або знижений операційною системою. Спочатку потік отримує значення базового пріоритету з діапазону базового пріоритету процесу, в якому він був створений.

Рисунок 12.6 – Схема призначення пріоритетів у Windows NT

Нехай, наприклад, значення базового пріоритету деякого процесу дорівнює *К*. Тоді усі потоки даного процесу отримають базові пріоритети з діапазону [*К-2, К + 2*]. Звідси видно, що, змінюючи базовий пріоритет процесу, ОС може впливати на базові пріоритети його потоків.

У Windows NT з плином часу пріоритет потоку може відхилятися від базового пріоритету потоку, причому ці зміни можуть бути не пов'язані зі змінами базового пріоритету процесу. ОС може підвищувати пріоритет потоку (який у цьому випадку називається динамічним) в тих випадках, коли потік не повністю використав відведений йому квант, або знижувати пріоритет, якщо квант був використаний повністю. ОС нарощує пріоритет в залежності від того, якого типу подія не дала потоку повністю використати квант.

Зокрема, ОС підвищує пріоритет у більшій мірі потокам, які очікують введення з клавіатури (інтерактивним додаткам) і в меншій мірі потокам, які виконують дискові операції. Саме на основі динамічних пріоритетів здійснюється планування потоків. Початковою точкою відліку для динамічного пріоритету є значення базового пріоритету потоку. Значення динамічного пріоритету потоку обмежено знизу його базовим пріоритетом, верхнім же кордоном є нижня межа діапазону пріоритетів реального часу.

Існують два різновиди пріоритетного планування: обслуговування з ***відносними пріоритетами*** і обслуговування з ***абсолютними пріоритетами***.

В обох випадках вибір потоку на виконання з черги готових здійснюється однаково: вибирається потік, що має найвищий пріоритет. Однак зміни активного потоку вирішується по-різному. У системах з відносними пріоритетами активний потік виконується до тих пір, поки він сам не покине процесор, перейшовши в стан очікування (або ж станеться помилка, або потік завершиться). На рис. 12.7, *а* показаний граф станів потоку в системі з відносними пріоритетами.

У системах з абсолютними пріоритетами виконання активного потоку переривається крім зазначених вище причин, ще за однієї умови: якщо в черзі готових потоків з'явився потік, пріоритет якого вище пріоритету активного потоку. Перерваний потік переходить в стан готовності (рис. 12.7, *б*).

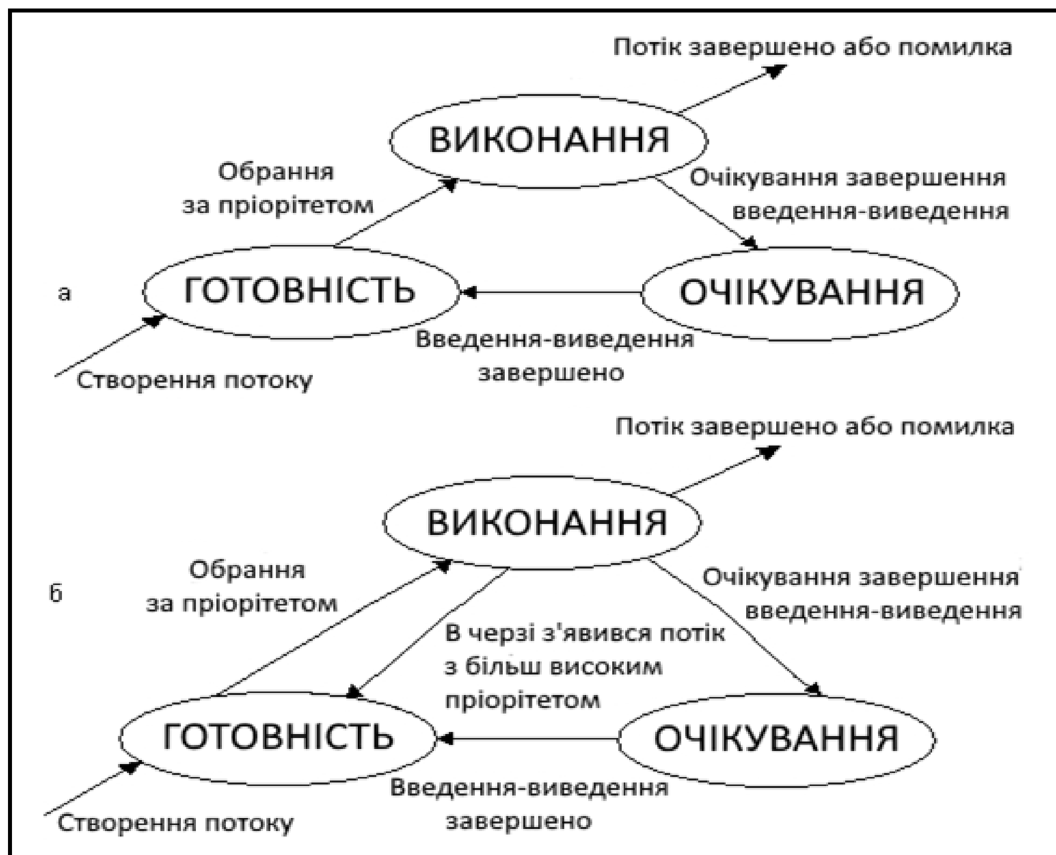


Рисунок 12.7 – Графи станів потоків в системах з відносними (а) і абсолютними пріоритетами (б)

У системах, в яких планування здійснюється на основі відносних пріоритетів, мінімізуються витрати на перемикання процесора з однієї роботи на іншу. З іншого боку, тут можуть виникати ситуації, коли одна задача займає процесор довгий час. Ясно, що для систем поділу часу і реального часу така дисципліна обслуговування не підходить, інтерактивний додаток може чекати своєї черги годинами, поки обчислювальної задачі не потрібно введення-виведення. А ось в системах пакетної обробки (в тому числі у відомої ОС OS/360) відносні пріоритети використовуються широко.

У системах з абсолютними пріоритетами час очікування потоку в чергах може бути зведений до мінімуму, якщо йому призначити найвищий пріоритет. Такий потік буде витісняти з процесора всі інші потоки (крім потоків, що мають такий же найвищий пріоритет). Це робить планування на основі абсолютних пріоритетів відповідним для систем управління об'єктами, в яких важлива швидка реакція на подію.

12.4 Змішані алгоритми планування

У багатьох операційних системах застосовуються змішані алгоритми планування з використанням як концепції квантування, так і пріоритетів.

Дисципліна обслуговування з пріоритетом, який залежить від часу обслуговування. Дисципліна обслуговування, заснована на абсолютних

пріоритетах. Під час виконання процесу його пріоритет зменшується з кожним тіком. Якщо пріоритет процесу стає менше пріоритету процесу, що стоїть в черзі готових, процес буде витіснений з виконання. Це дозволяє зменшити дискримінацію процесів, що виникає при використанні дисциплін обслуговування з абсолютними пріоритетами.

Схема дисципліни обслуговування з пріоритетами, залежними від часу виконання (а), і граф станів процесу (б) в системі з відповідною дисципліною обслуговування представлені на рис 12.8.

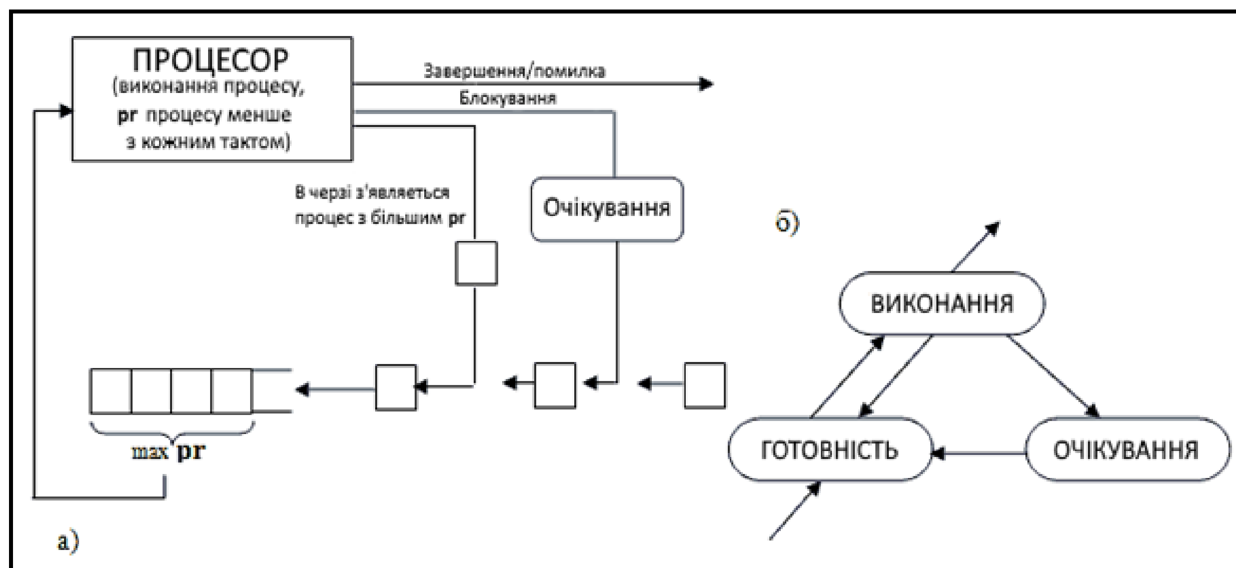


Рисунок 12.8 – Схема дисципліни обслуговування з пріоритетами, залежними від часу виконання (а)

Зміна завдання для виконування відбувається в таких випадках:
– процес завершений або сталася помилка;
– процес перейшов у стан очікування;
– пріоритет завдання стає менше, ніж у завдання з найбільшим пріоритетом у черзі готових завдань.

Переваги такої дисципліни обслуговування:
– враховується пріоритетність завдань;
– зменшується можливість недобросовісного використання механізмів пріоритетів.

Недоліки:
– можливість нескінченного відкладання фонових процесів;
– складна організація, так як необхідний перерахунок пріоритетів.

Дисципліна обслуговування з пріоритетом, який залежить від часу очікування в черзі готових процесів. Дисципліна обслуговування, заснована на абсолютних пріоритетах. У міру очікування в черзі готових, пріоритет процесу збільшується з кожним тіком. Якщо пріоритет процесу, який виконується, стає менше пріоритету процесу, який стоїть в черзі готових (незалежно від того, новий це процес або давно стоїть), процес буде витіснений з виконання. Це дозволяє виключити дискримінацію процесів, що виникає при

використанні дисциплін обслуговування з абсолютними пріоритетами і дисциплін обслуговування з пріоритетами, залежними від часу виконання.

Зміна завдання для виконання відбувається в таких випадках:

– процес завершений або сталася помилка;
– процес перейшов в стан очікування;
– пріоритет виконуванного завдання стає меншим, ніж в очікуючих у черзі готових завдань з найбільшим пріоритетом;
– в черзі з'явився процес з великим пріоритетом.

Схема дисципліни обслуговування з пріоритетами, залежними від часу очікування (а), і граф станів процесу (б) в системі з відповідною дисципліною обслуговування представлені на рис 12.9.

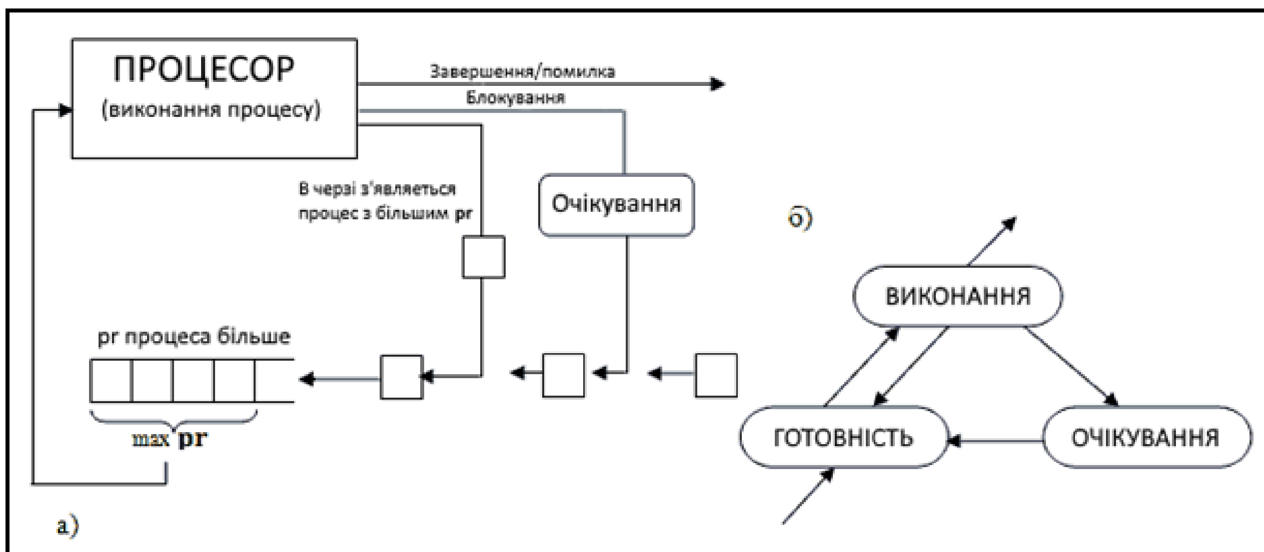


Рисунок 12.9 – Схема дисципліни обслуговування з пріоритетами, залежними від часу очікування (а)

12.5 Альтернативні стратегії планування

У табл. 12.1 представлена інформація про різні стратегії планування, які будуть розглянуті нижче.

Функція вибору процесу визначає, який з готових до виконання процесів буде обраний наступним для виконання. Функція може бути заснована на пріоритеті, вимогах до ресурсів або характеристиках виконання процесів. При цьому мають значення такі параметри:

– ***w*** – час, витрачений на цей момент системою (очікування і виконання);
– ***e*** – час, витрачений на цей момент на виконання;
– ***s*** – загальний час обслуговування, що потрібний процесу, включаючи *e* (зазвичай ця величина оцінюється або задається користувачем).

Наприклад, вибір функцій *max[w]* визначає стратегію «першим прийшов, першим обслужений» (first come first served – FCFS).

Режим рішення визначає, в які моменти часу виконується функція вибору. Режими рішення діляться на дві категорії.

1. **Невитісняючі**. У цьому випадку процес, який знаходиться в стані виконання, продовжує виконання до тих пір, поки він не завершиться або не опиниться в заблокованому стані.
2. **Витісняючі**. Процес, що виконується в даний момент, може бути перерваний і переведений ОС в стан готовності до виконання. Рішення про витіснення може прийматися при запуску нового процесу по перериванню, яке призводить заблокований процес в стан готовності, або періодично – на основі переривань таймера.

Таблиця 12.1. Характеристики різних стратегій планування

Стратегія планування	**Функція вибору**	**Режими виконання**	**Пропускна здатність**	**Голодування**
FCFS	max[w]	Невитісняючі	Не важливо	Немає
Кругова (RR)	const	Витісняючі (за часом)	Низька при малому кванті часу	Немає
SPN	min[s]	Невитісняючі	Висока	Можливо
SRT	min[s-e]	Витісняючі (за рішенням)	Висока	Можливо
HRRN	max[(w+s)/s]	Невитісняючі	Не важливо	Немає

Витісняючі стратегії призводять до підвищених накладних витрат у порівнянні з невитісняючими, але при цьому забезпечують кращий рівень обслуговування всієї множини процесів, оскільки запобігають монопольному використанню процесора протягом тривалого часу одним з процесів. Крім того, використання ефективних механізмів перемикання процесів і великий обсяг основної пам'яті дозволяє підтримувати відносно невелику вартість витіснення.

Розглянемо ці алгоритми планування процесів.

Першим поступив – першим обслужений (FCFS). Найпростіша стратегія планування «першим прийшов, першим обслужений» (first come first served – FCFS) відома також як схема строгої черговості. Як тільки процес стає готовим до виконання, він приєднується до черги готових процесів. При припиненні виконання поточного процесу для виконання вибирається процес, що знаходиться в черзі довше інших. Черга подібного типу має в програмуванні спеціальне найменування – FIFO (скорочення від First In, First Out – першим увійшов, першим вийшов). Такий алгоритм вибору процесу здійснює невитісняюче планування (рис. 12.10).

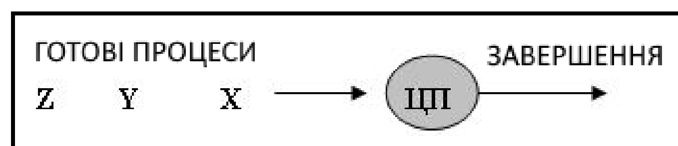


Рисунок 12.10 – Планування за принципом FIFO

Перевагою алгоритму FCFS є легкість його реалізації, але в той же час він має багато недоліків. Стратегія FCFS набагато краще працює для довгих

процесів, ніж для коротких. При роботі з процесами, орієнтованими на роботу з процесором, такі процеси отримують перевагу над процесами, орієнтованими на введення-виведення. Для однопроцесорних систем FCFS – це не найкраща стратегія, але вона часто комбінується з використанням пріоритетів. В цьому випадку планувальник підтримує ряд черг, по одній для кожного рівня пріоритету, і працює з процесами в кожній черзі за стратегією FCFS.

Розглянемо наступний приклад. Нехай в стані готовність знаходяться п'ять процесів (A, B, C, D, E), для яких відомий час появи в черзі готових процесів (0, 2, 4, 6, 8) і час їх виконання (обслуговування, CPU – 3, 6, 4, 5, 2). Ці часи наведені в деяких умовних одиницях.

Будемо вважати, що вся діяльність процесів обмежується використанням лише одного проміжку CPU, що процеси не роблять операцій введення-виведення, не будемо брати також до уваги і час перемикання контексту. Якщо процеси розташовані в черзі процесів, готових до виконання, в порядку A, B, C, D, E, то картина їх виконання виглядає таким чином (рис. 12.11).

Процес	A	B	C	D	E	Середній час
Час появи в черзі	0	2	4	6	8	
Час виконання	3	6	4	5	2	
Час завершення	3	9	13	18	20	
Час обороту	3	7	9	12	12	8.60
Час очікування	0	1	5	7	10	4.60
Нормалізований повний час	1.00	1.17	2.25	2.40	6.00	2.56

Рисунок 12.11 – Виконання процесів алгоритмом FCFS

Час очікування для процесу A становить 0 одиниць часу, для процесу B – 1, для процесу C – 5, для процесу D – 7, для процесу E – 10. Таким чином, ***середній час очікування*** в цьому випадку – (0 + 1 + 5 + 7 + 10)/5 = 4.60 одиниць часу.

Повний час виконання (час обороту), витрачений процесом в системі (час очікування + час обслуговування), для процесу A становить 3 одиниці часу, для процесу B – 1 + 6 = 7 одиниць, для процесу C – 5 + 4 = 9 одиниць, для процесу D – 7 + 5 = 12 одиниць, для процесу E – 10 + 2 = 12 одиниць. ***Середній повний час виконання*** виявляється рівним (3 + 7 + 9 + 12 + 12)/5 = 8.60 одиницям часу.

Кориснішою величиною є ***нормалізований повний час***, який визначається як відношення повного часу до часу виконання і вказує відносну затримку, що випробовується процесом. Мінімальне значення цього відношення – 1. Зростання значення відповідає зниженню рівня обслуговування.

На рис. 12.11 видно, що алгоритм FCFS набагато краще працює для довгих процесів, чим для коротких. Адже для коротких процесів час очікування може істотно перевищувати час обслуговування.

Кругове планування. Очевидний шлях підвищення ефективності роботи з короткими процесами в схемі FCFS – використання витіснення на основі таймера. Найпростіша стратегія, заснована на цій ідеї, – стратегія кругового (карусельного) планування (round robin – **RR**). Таймер генерує переривання через певні інтервали часу. При кожному перериванні процес, що виконується в даний момент, поміщається в чергу готових до виконання, і починає виконуватися черговий процес, який обирається відповідно до стратегії FCFS. Ця методика відома також як квантування часу, оскільки перед тим як опинитися витісненим, кожен процес отримує квант часу для виконання (рис. 12.12).

Рисунок 12.12 – Планування за принципом RR

При круговому плануванні важливим є питання про тривалість кванта часу. Так короткі процеси будуть відносно швидко проходити через систему, але з істотними накладними витратами, пов'язаними з обробкою переривань і виконанням функцій планувальника. Кругова стратегія ефективна в системах загального призначення з розподілом часу.

Таймер генерує переривання через певні інтервали часу. При кожному перериванні процес, що виконується, поміщається в чергу готових до виконання процесів, і починає виконуватися черговий процес, вибираний відповідно до стратегії FCFS. Ця методика відома також як «***квантування часу***» (time slicing), оскільки кожен процес отримує кванту часу для виконання (рис. 12.13) [22].

При круговому плануванні принциповим стає питання про тривалість кванта часу. При дуже великих величинах кванта часу, коли кожен процес встигає завершити свою роботу до виникнення переривання за часом, алгоритм RR вироджується в алгоритм FCFS. При дуже малих величинах створюється ілюзія того, що кожен з *n* процесів працює на власному віртуальному процесорі з продуктивністю ~ *1/n* від продуктивності реального процесора. Правда, це справедливо лише при теоретичному аналізі за умови нехтування часом перемикання контексту процесів. У реальних умовах при занадто малій величині кванта часу і, відповідно, занадто частому перемиканні контексту накладні витрати на перемикання різко знижують продуктивність системи.

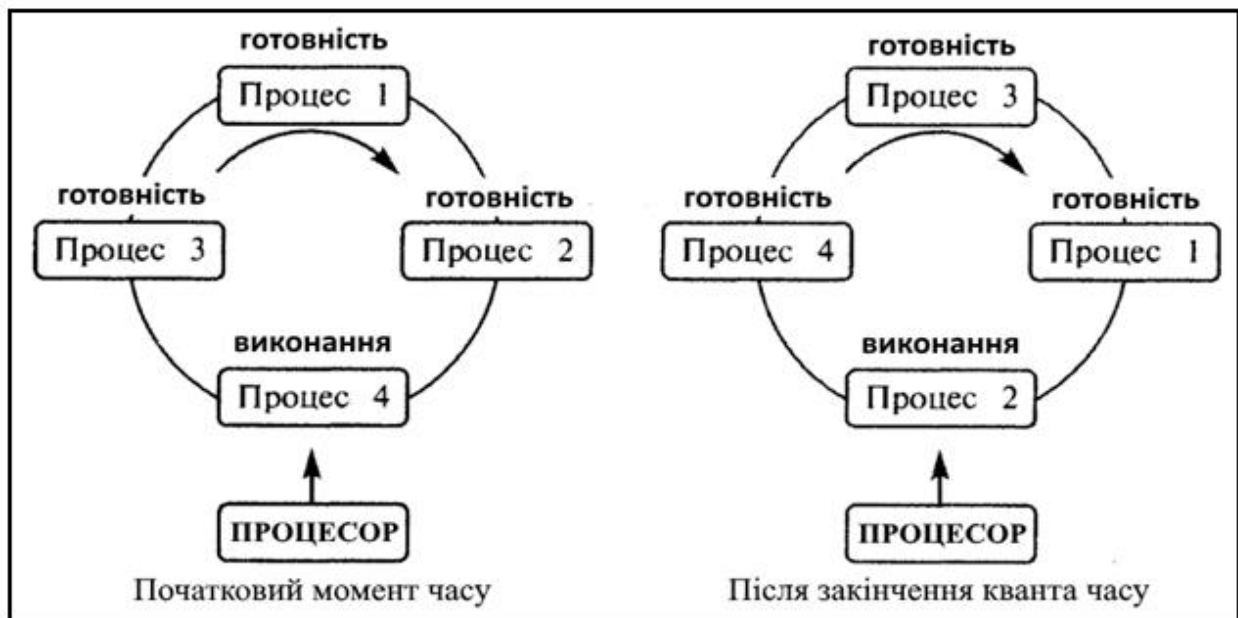
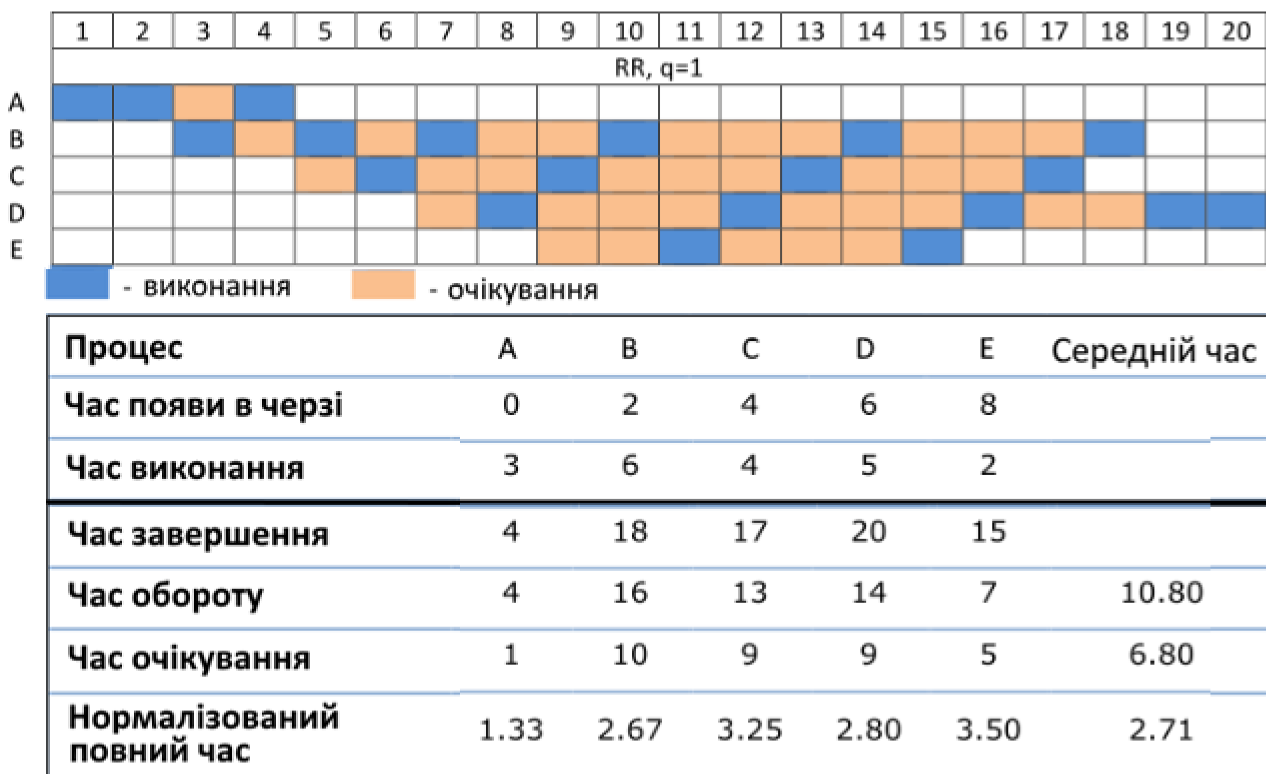


Рисунок 12.13 – Кругове планування

Розглянемо попередній приклад з тим же порядком процесів, часом появи в черзі готових процесів, часом їх виконання і величиною кванта часу q=1. Виконання цих процесів ілюструється рис. 12.14.

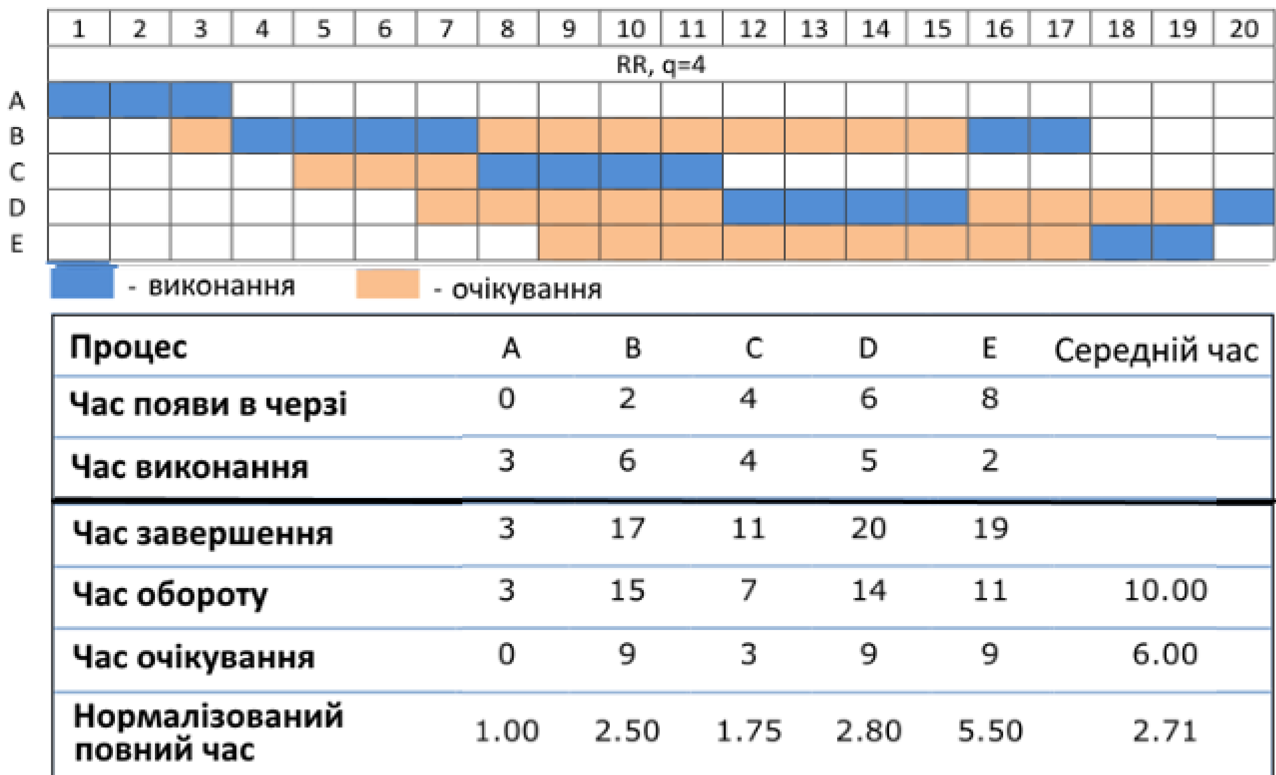


Процес	A	B	C	D	E	Середній час
Час появи в черзі	0	2	4	6	8	
Час виконання	3	6	4	5	2	
Час завершення	4	18	17	20	15	
Час обороту	4	16	13	14	7	10.80
Час очікування	1	10	9	9	5	6.80
Нормалізований повний час	1.33	2.67	3.25	2.80	3.50	2.71

Рисунок 12.14 – Виконання процесів алгоритмом RR з квантом часу q=1

На рис. 12.15 показані результати роботи RR алгоритму при використанні кванта часу q з тривалістю 4 одиниці часу.

Зверніть увагу, що найбільш короткий процес E значно швидше проходить через систему при малому кванті часу.



Процес	A	B	C	D	E	Середній час
Час появи в черзі	0	2	4	6	8	
Час виконання	3	6	4	5	2	
Час завершення	3	17	11	20	19	
Час обороту	3	15	7	14	11	10.00
Час очікування	0	9	3	9	9	6.00
Нормалізований повний час	1.00	2.50	1.75	2.80	5.50	2.71

Рисунок 12.15 – Виконання процесів алгоритмом RR з квантом часу q=4

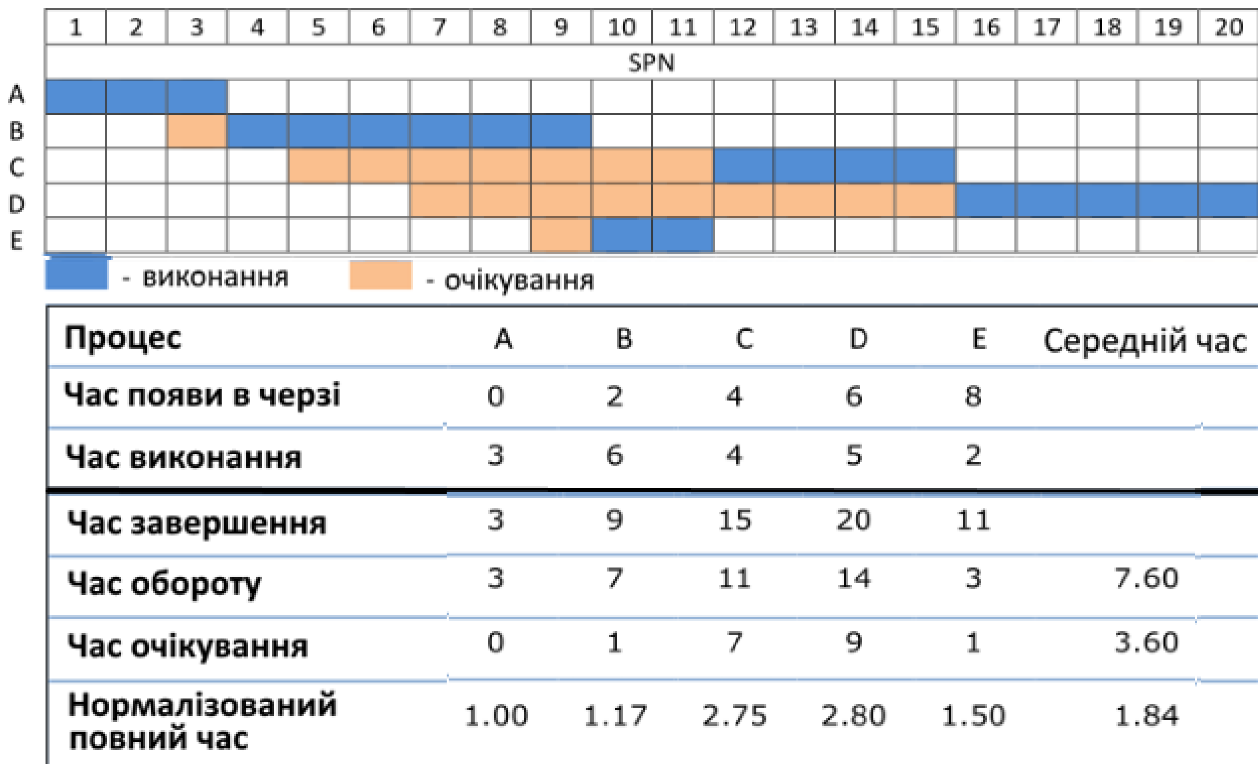
Вибір найкоротшого процесу. Ще один шлях до зниження перекосу на користь довгих процесів – використання стратегії вибору найкоротшого процесу (shortest process next - SPN). У літературі зустрічається і інша назва: «найкоротша робота першою» або Shortest Job First (**SJF**). Це не витісняюча стратегія, при якій для виконання вибирається процес з найменшим очікуваним часом виконання.

Основні труднощі в застосуванні стратегії SPN полягають у тому, як оцінити час виконання кожному процесу. Для пакетних завдань це може зробити програміст, а для виконання інтерактивних процесів операційна система підраховує час виконання за спеціальними формулами.

Основний ризик при використанні стратегії SPN полягає в можливому голодуванні довгих процесів при стабільній роботі коротких процесів. Його застосування небажане в системах з розподілом часу або системах обробки транзакцій через відсутність витіснення.

На рис. 12.16 приведені результати застосування цього алгоритму до нашого прикладу. Зверніть увагу, що процес Е обслуговується набагато раніше, ніж у разі застосування алгоритму FCFS. Відносно часу відгуку загальна продуктивність системи також зростає, але при цьому збільшується розкид його величини, особливо для довгих процесів, і, відповідно, знижується передбачуваність.

Основний ризик при використанні алгоритму SPN полягає в можливому голодуванні довгих процесів при стабільній роботі коротких процесів.



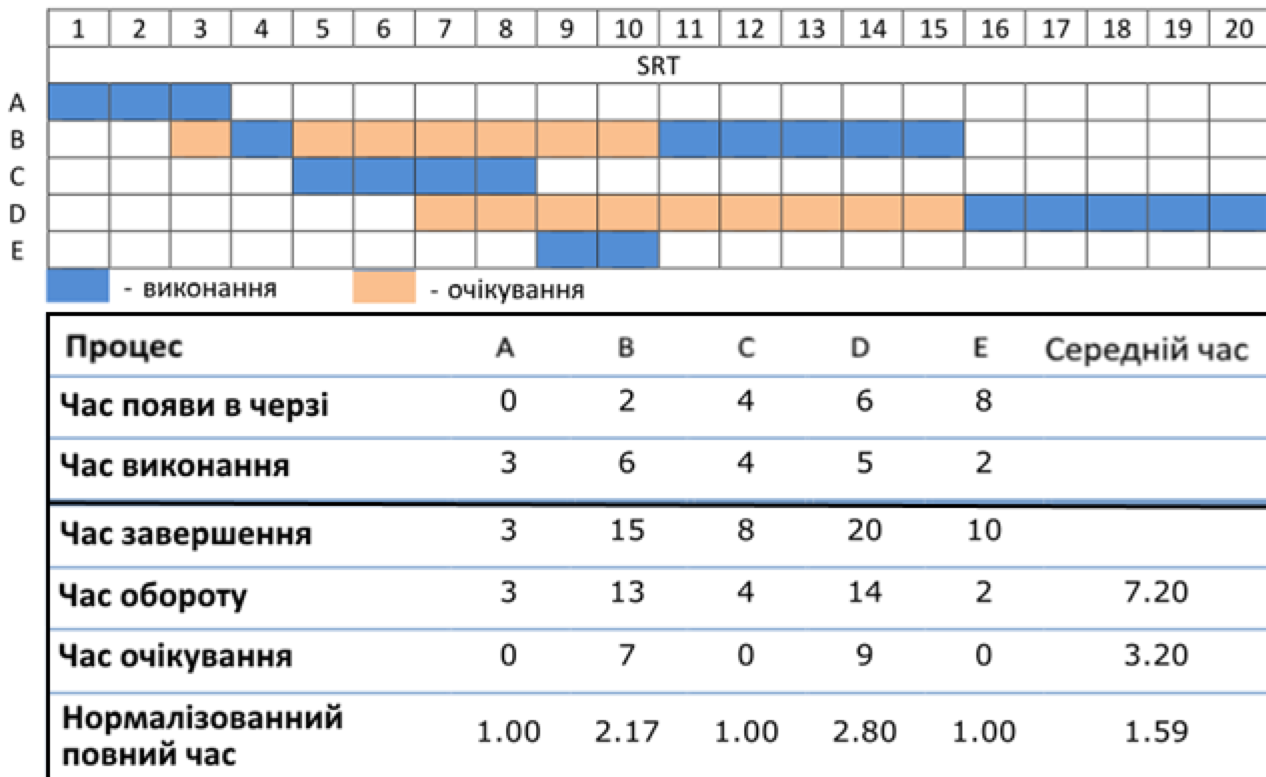
Процес	A	B	C	D	E	Середній час
Час появи в черзі	0	2	4	6	8	
Час виконання	3	6	4	5	2	
Час завершення	3	9	15	20	11	
Час обороту	3	7	11	14	3	7.60
Час очікування	0	1	7	9	1	3.60
Нормалізований повний час	1.00	1.17	2.75	2.80	1.50	1.84

Рисунок 12.16 – Виконання процесів алгоритмом SPN

Найменший час, що залишається. Стратегія найменшого часу, що залишається (shortest remaining time – **SRT**) являє собою витісняючу версію стратегії SPN. У цьому випадку планувальник вибирає процес з найменшим очікуваним часом до закінчення процесу. При приєднанні нового процесу до черги готових до виконання процесів може виявитися, що його час, що залишився, менше, ніж час, що залишився у виконуваного в даний момент процесу. Планувальник, відповідно, може застосувати витіснення при готовності нового процесу. Як і при використанні стратегії SPN, планувальник для коректної роботи функції вибору повинен оцінити час виконання процесу. У цьому випадку також є ризик голодування довгих процесів.

У разі використання алгоритму SRT немає таких великих перекосів на користь довгих процесів, як при використанні алгоритму FCFS. У відмінності від алгоритму RR, тут не генеруються додаткові переривання, що знижує накладні витрати. Проте, в цьому випадку відбувається збільшення накладних витрат із-за необхідності фіксувати і записувати час виконання процесів. У зв'язку з тим що короткі завдання негайно отримують перевагу перед довгими завданнями, що виконуються, алгоритм SRT істотно виграє в алгоритму SPN в часі обороту. На рис. 12.17 приведені результати застосування цього алгоритму до нашого прикладу.

В даному прикладі три найбільш коротких процеси обслуговуються негайно, що призводить до нормалізованого часу обороту для кожного з них, рівному 1.00.



Процес	A	B	C	D	E	Середній час
Час появи в черзі	0	2	4	6	8	
Час виконання	3	6	4	5	2	
Час завершення	3	15	8	20	10	
Час обороту	3	13	4	14	2	7.20
Час очікування	0	7	0	9	0	3.20
Нормалізований повний час	1.00	2.17	1.00	2.80	1.00	1.59

Рисунок 12.17 – Виконання процесів алгоритмом SRT

Планування за найбільшим відносним часом реакції. Брінч Хансен розробив стратегію планування за найбільшим відносним часом реакції (highest response ratio next – **HRRN**), який компенсує деякі недоліки алгоритму SPF (найкоротший процес першим): надмірне упередження проти довгих процесів і надмірну прихильність до коротких нових процесів. HRRN – це дисципліна планування без пріоритетного витіснення, згідно з якою пріоритет кожного процесу є не тільки функцією часу обслуговування цього процесу, але також часу, витраченого процесом на очікування обслуговування. Після того як процес отримає у своє розпорядження процесор, він виконується до завершення. Динамічні пріоритети при дисципліні обслуговування HRRN обчислюються за формулою:

$$P = (W + S)/S,$$

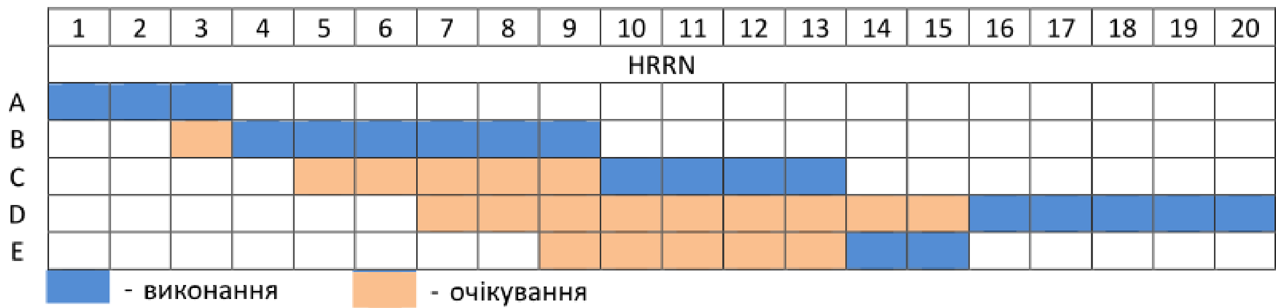
де P – пріоритет;

W – час, витрачений процесом на очікування;

S – очікуваний час обслуговування.

Зауважимо, що процес при вході в систему отримує мінімальне значення P (рівне 1.0). Таким чином, правило стратегії планування найвищого відносного часу реакції (відгуку) можна сформулювати так: при завершенні або блокуванні поточного процесу для виконання з черги готових процесів вибирається той, який має найбільше значення P.

Приклад роботи алгоритму HRRN представлений на рис. 12.18. Як і у разі алгоритмів SRT і SPN, в описаному алгоритмі потрібна оцінка часу обслуговування для визначення максимального значення P.



Процес	A	B	C	D	E	Середній час
Час появи в черзі	0	2	4	6	8	
Час виконання	3	6	4	5	2	
Час завершення	3	9	13	20	15	
Час обороту	3	7	9	14	7	8.00
Час очікування	0	1	5	9	5	4.00
Нормалізований повний час	1.00	1.17	2.25	2.80	3.50	2.14

Рисунок 12.18 – Виконання процесів алгоритмом HRRN

Оскільки час виконання *S* знаходиться в знаменнику, та перевага виявлятиметься коротшим процесам. Проте, оскільки в чисельнику є час очікування *W*, довші процеси, які вже досить довго чекають, також отримуватимуть певну перевагу. На практиці пріоритети перераховуються через певні проміжки часу, і відповідно до змін черга переупорядковується.

12.6 Планування в Windows 2000

12.6.1 Процеси і потоки

У різних ОС процеси реалізуються по-різному. Ці відмінності полягають у тому, якими структурами даних представлені процеси, як вони іменуються, якими способами захищені один від одного і які стосунки існують між ними. Процеси Windows 2000 (W2K) мають такі характерні властивості:

1. Процеси W2K реалізовані у формі об'єктів, і доступ до них здійснюється за допомогою служби об'єктів.
2. Процес W2K має багатопотокову організацію.
3. Як об'єкти-процеси, так і об'єкти-потоки мають вбудовані засоби синхронізації.
4. Менеджер процесів W2K не підтримує між процесами стосунків типу «батько-нащадок».

У будь-якій системі поняття «процес» включає:

– виконуваний код;

- власний адресний простір, який є сукупністю віртуальних адрес, які може використати процес;
- ресурси системи, такі як файли, семафори тощо, які призначені процесу операційною системою;
- хоч би один виконуваний потік.

Адресний простір кожного процесу захищений від втручання в нього будь-якого іншого процесу. Це забезпечується механізмами віртуальної пам'яті. Операційна система, звичайно, теж захищена від прикладних процесів. Щоб виконати яку-небудь процедуру ОС або прочитати що-небудь з її області пам'яті, потік повинен виконуватися в режимі ядра. Призначені для користувача процеси отримують доступ до функцій ядра за допомогою системних викликів. У призначеному для користувача режимі виконуються не лише прикладні програми, але й захищені підсистеми W2K.

У W2K процес – це просто об'єкт, що створюється і знищується менеджером об'єктів. Об'єкт-процес, як і інші об'єкти, містить заголовок, який створює і ініціалізує менеджер об'єктів. Менеджер процесів визначає атрибути, що зберігаються в тілі об'єкта-процесу, а також забезпечує системний сервіс, який відновлює і змінює ці атрибути.

До числа атрибутів тіла об'єкта-процесу входять:

1. Ідентифікатор процесу – унікальне значення, яке ідентифікує процес у рамках операційної системи.
2. Ознака (token – токен) доступу – виконуваний об'єкт, що містить інформацію про безпеку.
3. Базовий пріоритет – основа для виконавчого пріоритету потоку процесу.
4. Процесорна сумісність – набір процесорів, на яких можуть виконуватися потоки процесу.
5. Граничні значення квот – максимальна кількість сторінкової і несторінкової системної пам'яті, дискового простору, призначеного для вивантаження сторінок, процесорного часу, які можуть бути використані процесами користувача.
6. Час виконання – загальна кількість часу, впродовж якого виконуються усі потоки процесу.

Нагадаємо, що потік є виконуваною одиницею, яка розташовується в адресному просторі процесу і використовує ресурси, виділені процесу. Подібно до процесу потік в W2K реалізований у формі об'єкта і управляється менеджером об'єктів.

Об'єкт-потік має такі атрибути:

1. Ідентифікатор клієнта – унікальне значення, яке ідентифікує потік при його зверненні до сервера.
2. Контекст потоку – інформація, яка потрібна ОС для того щоб продовжити виконання перерваного потоку. Контекст потоку містить поточний стан регістрів, стеків і індивідуальної області пам'яті, яка використовується підсистемами і бібліотеками.
3. Динамічний пріоритет – значення пріоритету потоку в даний момент.
4. Базовий пріоритет – нижня межа динамічного пріоритету потоку.

5. Процесорна сумісність потоків – перелік типів процесорів, на яких може виконуватися потік.
6. Час виконання потоку – сумарний час виконання потоку в призначеному для користувача режимі і в режимі ядра, накопичений за період існування потоку.
7. Стан попередження – прапор, який показує, що потік повинна виконувати виклик асинхронної процедури.
8. Лічильник призупинень – поточна кількість призупинень виконання потоку.

Окрім перерахованих атрибутів, є і деякі інші атрибути. Як видно з переліку, багато атрибутів об'єкта-потоку аналогічні атрибутам об'єкта-процесу. Схожі і сервісні функції, які можуть бути виконані над об'єктами-процесами і об'єктами-потоками: створення, відкриття, завершення, призупинення, запит і установка інформації, запит і установка контексту та інші функції.

12.6.2 Планування процесів і потоків

Планування в Windows здійснюється на рівні потоків, а не процесів. Це здається зрозумілим, оскільки самі процеси не виконуються, а лише надають ресурси і контекст для виконання потоків. Тому при плануванні потоків система не звертає уваги на те, якому процесу вони належать. Наприклад, якщо процес А має 10 готових до виконання потоків, а процес В – два, і усі 12 потоків мають однаковий пріоритет, кожен з потоків отримає 1/12 процесорного часу.

У W2K реалізована витісняюча багатозадачність, при якій операційна система не чекає, коли потік сам захоче звільнити процесор, а примусово знімає його з виконання після того, як той витратив відведений йому час (квант), або якщо в черзі готових з'явився потік з вищим пріоритетом. При такій організації розподілу процесора жоден потік не займе процесор на дуже довгий час. В ОС W2K потік у ході свого існування може мати один з шести станів (рис. 12.19).

Життєвий цикл потоку починається в той момент, коли програма створює новий потік. Менеджер процесів виділяє пам'ять для об'єкта-потоку і звертається до ядра, щоб ініціалізувати об'єкт-потік ядра. Після ініціалізації потік проходить через наведені нижче стани:

Готовність. При пошуку потоку на виконання диспетчер переглядає тільки потоки, що знаходяться в стані готовності, в яких є все для виконання, але бракує тільки процесора.

Першочергова готовність (резервний). Для кожного процесора системи вибирається один потік, який виконуватиметься наступним (найперший потік в черзі). Коли умови дозволяють, відбувається перемикання на контекст цього потоку.

Виконання. Як тільки відбувається перемикання контекстів, потік переходить в стан виконання і знаходиться в ньому до тих пір, поки або ядро не витіснить його через те, що з'явився пріоритетніший потік або закінчився квант часу, виділений цьому потоку, або потік завершиться взагалі, або він за власною ініціативою перейде в стан очікування.

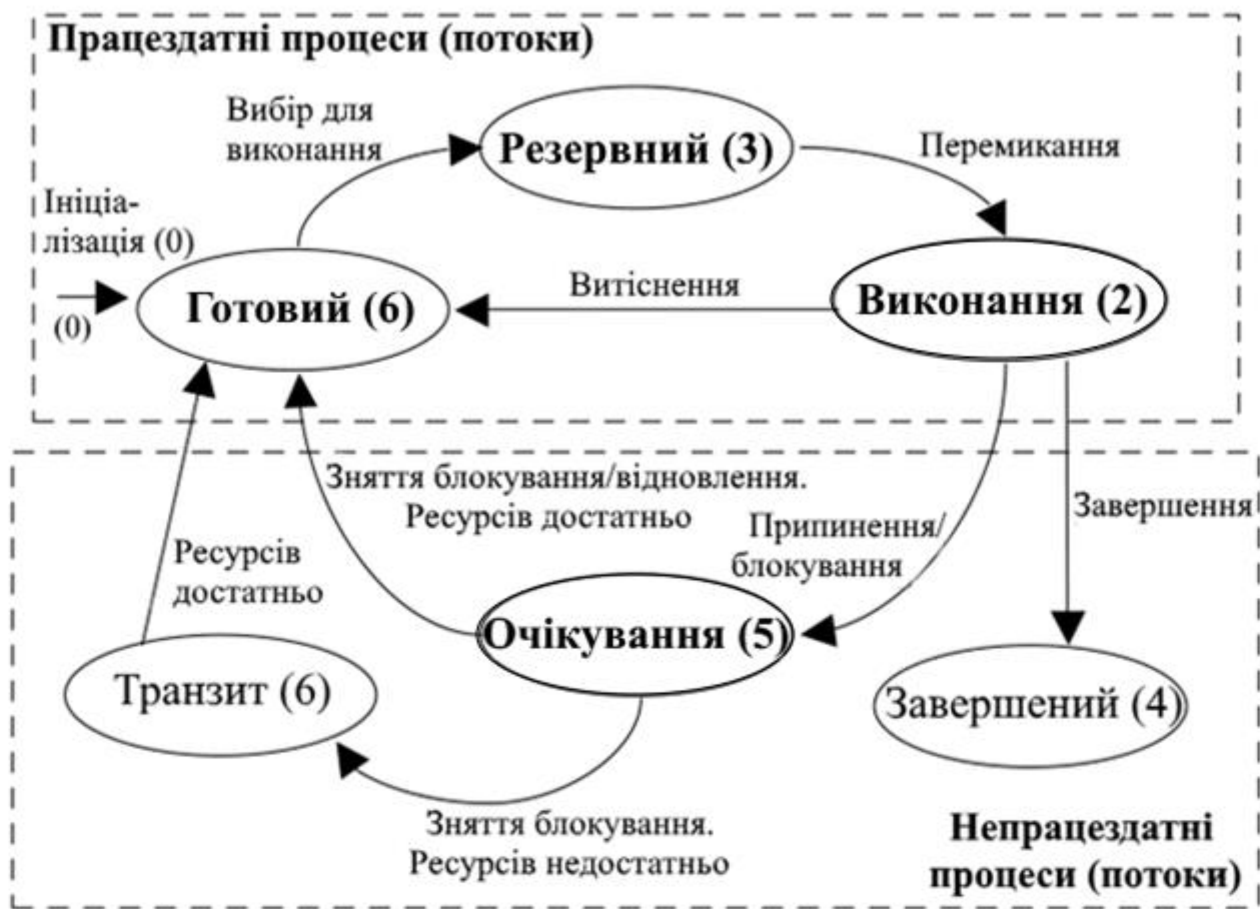


Рисунок 12.19 – Граф станів потоків в Windows

Очікування. Потік може входити в стан очікування декількома способами: потік за своєю ініціативою чекає деякий об'єкт для того, щоб синхронізувати своє виконання; операційна система (наприклад, підсистема введення-виведення) може чекати в інтересах потоку; підсистема оточення може безпосередньо змусити потік призупинити себе. Коли очікування потоку добіжить кінця, він повертається в стан готовності.

Перехідний стан (транзит). Потік входить в перехідний стан, якщо він готовий до виконання, але ресурси, які йому потрібні, зайняті. Наприклад, сторінка, що містить стек потоку, може бути вивантажена з оперативної пам'яті на диск. При звільненні ресурсів потік переходить в стан готовності.

Завершення. Коли виконання потоку закінчилося, він входить в стан завершення. Знаходячись в цьому стані, потік може бути або видалений, або не видалений. Це залежить від алгоритму роботи менеджера об'єктів, відповідно до якого він і вирішує, коли видаляти цей об'єкт.

Для визначення порядку виконання потоків диспетчер ядра використовує алгоритм, заснований на пріоритетах, відповідно до якого кожному потоку привласнюється число – пріоритет, і потоки з вищим пріоритетом виконуються раніше потоків з нижчим пріоритетом.

На початку потік отримує пріоритет від процесу, який його створює. У свою чергу, процес отримує пріоритет у той момент, коли його створює підсистема того або іншого прикладного середовища. Значення базового

пріоритету надається процесу системою за умовчанням або системним адміністратором. Потік наслідує цей базовий пріоритет і може змінити його, трохи збільшивши або зменшивши. У ході виконання пріоритет планування може мінятися.

У Windows 2000 пріоритети організовані у вигляді двох груп, або класів: реального часу і динамічні. Кожна з груп складається з 16 рівнів пріоритетів (рис. 12.20). Потоки, що вимагають негайної уваги, знаходяться в класі реального часу, який включає функції здійснення комунікацій і задачі реального часу. Інші потоки потрапляють в клас потоків зі змінними (призначеними для користувача) пріоритетами.

Кожного разу, коли необхідно вибрати потік для виконання, диспетчер переглядає чергу готових потоків реального часу і звертається до інших потоків тільки тоді, коли черга потоків реального часу порожня. Більшість потоків в системі потрапляють в клас потоків зі змінними пріоритетами, діапазон пріоритетів яких від 0 до 15. Цей клас має назву «***Змінні пріоритети***» (***динамічні пріоритети потоку***) тому, що диспетчер настроює систему, вибираючи (знижуючи або підвищуючи) пріоритети потоків цього класу.

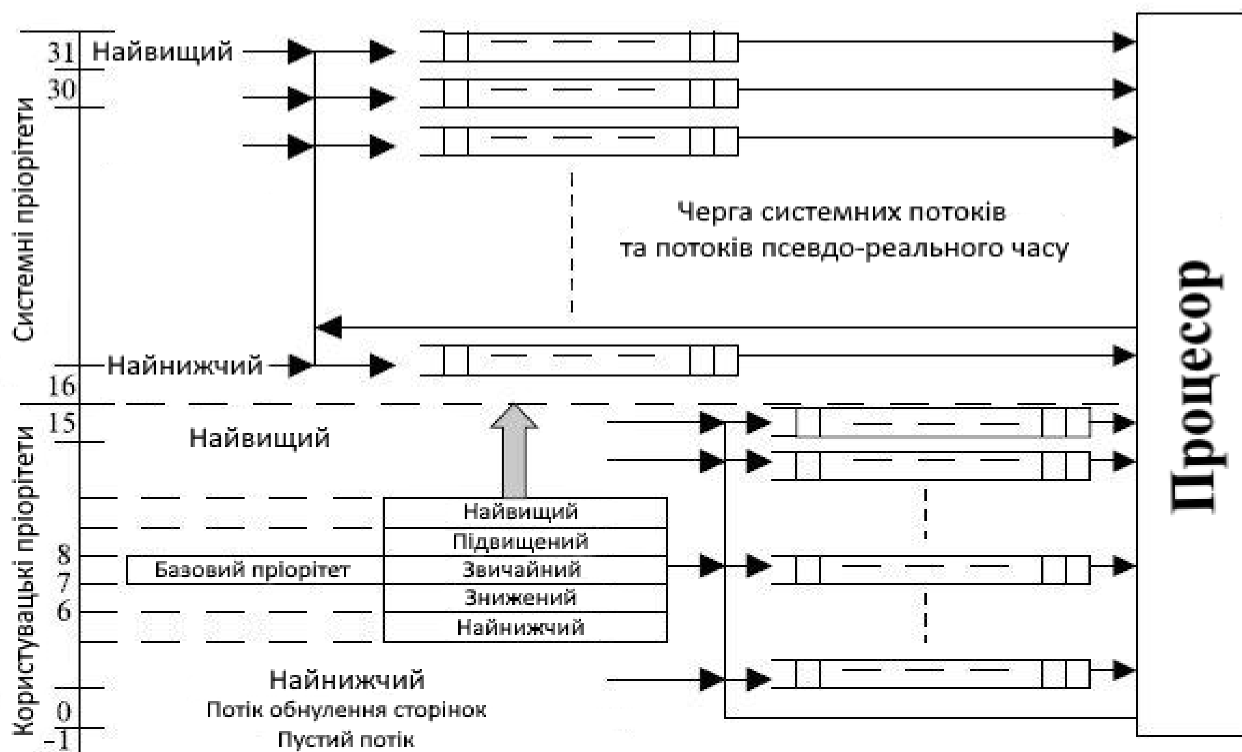


Рисунок 12.20 – Планування в Window

Алгоритм планування потоків в W2K об'єднує в собі обидві базові концепції – ***квантування*** і ***пріоритети.*** Як і в усіх інших алгоритмах, заснованих на квантуванні, кожному потоку надається квант часу, впродовж якого він може виконуватися.

Наприклад, для Win2000 Professional початкове значення кванта дорівнює 6, а для Win2000 Server – 36. Всякий раз, коли виникає переривання від таймера, із кванта потоку віднімається 3, і так до тих пір, поки він не досягне нуля. Частота

спрацьовування таймера залежить від апаратної платформи. Наприклад, для більшості однопроцесорних х86 систем він складає 10 мс, а на більшості багатопроцесорних х86 систем – 15 мс.

Потік звільняє процесор, якщо:

- блокується, йдучи в стан очікування;
- завершується;
- вичерпаний квант;
- у черзі готових потоків з'являється пріоритетніший потік.

Використання ***динамічних пріоритетів***, що змінюються в часі, дозволяє реалізувати адаптивне планування, при якому не дискримінуються інтерактивні задачі, що часто виконують операції введення-виведення, і які не використовують повністю виділені їм кванти часу. Якщо потік повністю вичерпав свій квант, то його пріоритет знижується на деяку величину. В той же час пріоритет потоків, які перейшли в стан очікування, не використавши повністю виділений їм квант часу, підвищується. Пріоритет не змінюється, якщо потік витіснений пріоритетнішим потоком.

Для того щоб забезпечити хороший час реакції системи, алгоритм планування використовує разом з квантуванням концепцію ***абсолютних пріоритетів***. Відповідно до цієї концепції при появі в черзі готових потоків, в яких пріоритет вищий, ніж у того, що виконується в даний момент, відбувається зміна активного потоку на потік з найвищим пріоритетом.

Прив'язка до процесорів. Якщо операційна система виконується на машині, де встановлено більше одного процесор, то за умовчанням потік виконується на будь-якому доступному процесорі. Проте в деяких випадках набір процесорів, на яких потік може виконуватись, може бути обмежений. Це явище називається прив'язкою до процесорів (processor affinity). Можна змінити прив'язку до процесорів програмно, через Win32-функції планування.

У багатопроцесорній системі з *N* процесорами завжди активні потоки з найвищими пріоритетами. З іншими потоками з нижчим пріоритетом *N-1* працює єдиний процесор, що залишився.

У багатопроцесорних системах при диспетчеризації і плануванні потоків основною характеристикою є їх процесорна сумісність. Після того як ядро вибрало потік з найвищим пріоритетом, воно перевіряє, який процесор може виконати цей потік. Якщо атрибут потоку «***процесорна сумісність***» не дозволяє потоку виконуватися ні на одному з вільних процесорів, то вибирається наступний в порядку пріоритетів потік.

При роботі W2K в системі з одним процесором потік з найвищим пріоритетом завжди активний, якщо тільки не чекає настання якої-небудь події. Якщо є декілька потоків з найвищим пріоритетом, то згідно з описаною схемою процесор працює з ними з використанням кругового планування.

Пам'ять. Кожному процесу в Win32 доступний лінійний 4-гігабайтний (2^{32} = 4 294 967 296) віртуальний адресний простір. Зазвичай верхня половина цього простору резервується за операційною системою, а друга половина доступна процесу.

Віртуальний адресний простір процесу доступний усім потокам цього процесу. Іншими словами, усі потоки одного процесу виконуються в єдиному адресному просторі. З іншого боку, механізм віртуальної пам'яті дозволяє ізолювати процеси один від одного. Потоки одного процесу не можуть посилатися на адресний простір іншого процесу.

Оскільки далеко не всякий комп'ютер в змозі виділити по 4 Гб фізичної пам'яті на кожен процес, використовується механізм підкачування (swapping). Коли оперативної пам'яті бракує, операційна система переміщає частину вмісту пам'яті на диск, у файл (swap file або page file), звільняючи, таким чином, фізичну пам'ять для інших процесів. Коли потік звертається до сторінки віртуальної пам'яті, записаної на диск, диспетчер віртуальної пам'яті завантажує цю інформацію з диска назад в пам'ять.

Створення процесів. Створення Win32 процесу здійснюється викликом однієї з таких функцій, як CreateProcess, CreateProcessAsUser (для Win NT/2000) і CreateProcessWithLogonW (починаючи з Win2000) і відбувається в декілька етапів.

1. Відкривається файл образу (.exe), який виконуватиметься в процесі. Якщо виконуваний файл не є Win32 додатком, то шукається образ підтримки (support image) для запуску цієї програми. Наприклад, якщо виконується файл з розширенням .bat, запускається cmd.exe тощо.
2. Створюється об'єкт Win32 «процес».
3. Створюється первинний потік (стек, контекст і об'єкт «потік»).
4. Підсистема Win32 повідомляється про створення нового процесу і потоку.
5. Починається виконання первинного потоку.
6. У контексті нового процесу і потоку ініціалізувався адресний простір (наприклад, завантажуються необхідні dll-файли) і починається виконання програми.

Завершення процесів. Процес завершується якщо:

- вхідна функція первинного потоку повернула управління;
- один з потоків процесу викликав функцію ExitProcess;
- потік іншого процесу викликав функцію TerminateProcess.

Створення потоків. Первинний потік створюється автоматично при створенні процесу. Інші потоки створюються функціями CreateThread і CreateRemoteThread (тільки у Win NT/2000/XP).

Завершення потоків. Потік завершується якщо:

- функція потоку повертає управління;
- потік самознищується, викликавши ExitThread;
- інший потік цього або стороннього процесу викликає TerminateThread;
- завершується процес, що містить цей потік.

Об'єкти ядра. Ці об'єкти використовуються системою і додатками користувача для управління різними ресурсами: процесами, потоками, файлами тощо. Windows дозволяє створювати і оперувати з декількома типами таких об'єктів.

Об'єкт ядра – це, по суті, структура, створена ядром і доступна тільки йому. У додаток користувача передається тільки описувач (handle) об'єкта, а управляти об'єктом ядра можна за допомогою функцій Win32 API.

Wait–функції. Роботу потоку можна призупинити. Для цього існує багато способів. Ось деякі з них.

Функція Sleep() призупиняє роботу потоку на задане число мілісекунд. Якщо в якості аргументу вказати 0 ms, то станеться наступне. Потік відмовиться від свого кванта процесорного часу, проте тут же з'явиться в списку потоків готових до виконання. Іншими словами станеться навмисне перемикання потоків, вірніше сказати, спроба перемикання. Адже наступним для виконання потоком цілком може стати той же самий потік.

Функція WaitForSingleObject() призупиняє виконання потоку до тих пір, поки не станеться одна з двох подій:

– закінчиться таймаут очікування;
– очікуваний об'єкт перейде в сигнальний стан.

За поверненим значенням можна зрозуміти, яка з двох подій сталася. Чекати за допомогою wait-функцій може більшість об'єктів ядра, наприклад, об'єкт «процес» або «потік», щоб визначити, коли вони завершать свою роботу.

Функції WaitForMultipleObjects передається відразу масив об'єктів. Можна чекати спрацьовування відразу всіх об'єктів або якогось одного з них.

12.6.3 Синхронізація потоків

Працюючи паралельно, потоки спільно використовують адресний простір процесу. Також усі вони мають доступ до описувачів відкритих у процесі об'єктів. Якщо декілька потоків одночасно звертаються до одного ресурсу або необхідно якось упорядкувати роботу потоків, то для цього використовують об'єкти синхронізації і відповідні механізми.

М'ютекси (Mutex) – це об'єкти ядра, які створюються функцією CreateMutex(). М'ютекс буває в двох станах – зайнятому і вільному. М'ютексом добре захищати одиничний ресурс від одночасного звернення до нього різних потоків.

Семафори (Semaphore) створюється функцією CreateSemaphore(). Він дуже схожий на м'ютекс, тільки на відміну від нього у семафора є лічильник. Семафор відкритий якщо лічильник більше 0 і закритий, якщо лічильник дорівнює 0.

Події (Event) як і м'ютекси мають два стани – встановлений і скинутий. Події бувають із скиданням вручну і з автоскиданням. Коли потік дочекався (wait-функція повернула управління) події з автоскиданням, така подія автоматично скидається. Інакше подію треба скидати вручну, викликавши функцію ResetEvent(). Припустимо, що відразу декілька потоків чекають однієї і тієї ж події, і подія спрацювала. Якщо це була подія з автоскиданням, то вона дозволить працювати тільки одному потоку (адже відразу ж після повернення з його wait-функції подія скинеться автоматично), а інші потоки залишаться чекати. Якщо ж це була подія із скиданням вручну, то усі потоки отримають

управління, а подія так і залишиться у встановленому стані, поки який-небудь потік не викличе ResetEvent().

Критичні секції. Синхронізація в режимі користувача. Критична секція гарантує вам, що спеціальні ділянки коду програми не виконуватимуться одночасно. Строго кажучи, критична секція не є об'єктом ядра. Вона є структурою, що містить декілька прапорів і якийсь (не важливо) об'єкт ядра. При вході в критичну секцію спочатку перевіряються прапори, і якщо з'ясовується, що вона вже зайнята іншим потоком, то виконується звичайна wait-функція. Критична секція примітна тим, що для перевірки, чи зайнята вона чи ні, програма не переходить в режим ядра (не виконується wait-функція), а лише перевіряються прапори. Через це вважається, що синхронізація за допомогою критичних секцій найшвидш а. Таку синхронізацію називають «синхронізація в режимі користувача».

Синхронізація процесів. Описувачі об'єктів ядра залежні від конкретного процесу. Існують способи роботи з одними і тими ж об'єктами ядра різними процесами.

По-перше, це спадок описувача. При створенні об'єкта можна вказати чи буде його описувач наслідувати дочірні (породжені цим процесом) процеси.

По-друге, дублювання описувача. Функція DuplicateHandle дублює описувач об'єкту одного процесу в іншій. Тобто, по суті, бере запис в таблиці описувачів одного процесу і створює його копію в таблиці іншого.

І, нарешті, іменування об'єкту ядра. При створенні об'єкта ядра для синхронізації (м’ютекса, семафора, очікуваного таймера або події) можна задати його ім'я. Воно має бути унікальним в системі. Тоді інший процес може відкрити цей об'єкт ядра, вказавши у функції Open(OpenMutex, OpenSemaphore, OpenWaitableTimer, OpenEvent) це ім'я. Насправді, при виклику функції Create...() система спочатку перевіряє, чи не існує вже об'єкт ядра з таким іменем. Якщо ні, то створюється новий об'єкт. Якщо так, ядро перевіряє тип цього об'єкту і права доступу. Якщо типи не співпадають або ж процес не має повних прав на доступ до об'єкта, виклик Create…() функції закінчується невдало і повертається NULL. Якщо все нормально, то просто створюється новий описувач (handle) існуючого вже об'єкту ядра. За кодом повернення функції GetLastError() можна зрозуміти, що сталося: створився новий об'єкт або Create() повернула вже існуючий.

Тому синхронізувати потоки всередині різних процесів можна так як і в межах одного. Треба тільки правильно передати описувач синхронізуючого об'єкта від одного процесу до іншого будь-яким з перелічених вище способів.

12.6.4 Взаємодія між процесами

Потоки одного процесу не мають доступу до адресного простору іншого процесу. Однак існують механізми для передачі даних між процесами.

Колективна пам'ять. Як уже говорилося, система віртуальної пам'яті в Win32 використовує файл підкачки – swap file (або файл розміщення – page file), маючи можливість перетворення сторінок оперативної пам'яті в сторінки файлу

на диску і навпаки. Система може проектувати на оперативну пам'ять не тільки файл розміщення, а й будь-який інший файл. Додатки можуть використовувати цю можливість. Це може використовуватися для забезпечення швидшого доступу до файлів, а також для спільного використання пам'яті.

Такі об'єкти називаються проекціями файлів на оперативну пам'ять (file-mapping object). Для створення проекції файлу спочатку викликається функція CreateFileMapping(). Їй передається дескриптор вже відкритого файлу або вказується, що потрібно використовувати page file операційної системи. Крім цього, в параметрах їй передається прапор захисту, максимальний розмір проекції і ім'я об'єкта. Потім викликається функція MapViewOfFile(). Вона відображає уявлення файлу (view of a file) в адресний простір процесу. Після закінчення роботи викликається функція UnmapViewOfFile(). Вона звільняє пам'ять і записує дані у файл (якщо це не файл підкачки). Щоб записати дані на диск негайно, використовується функція FlushViewOfFile(). Проекція файлу, як і інші об'єкти ядра, може використовуватися іншими процесами через успадкування, дублювання дескриптора або за іменем.

Інші механізми (сокети, англ. pipe**).** Крім колективної пам'яті, в Windows є й інші способи передачі інформації між процесами, наприклад, канали, іменовані канали та сокети. Усі вони мають подібний принцип і являють собою своєрідний канал або з'єднання, «трубу», що сполучає процеси. Програма, маючи один кінець такого з'єднання, може читати і/або писати в нього дані, обмінюючись інформацією з програмою на іншому кінці.

Канали використовуються для пересилання даних в одному напрямку між дочірнім та батьківським процесами або між двома дочірніми процесами. Операції читання/запису в канал схожі на подібні операції при роботі з файлами.

Іменовані канали використовуються для двостороннього обміну даними між процесом-сервером і одним або декількома процесами-клієнтами. Як і анонімні канали, вони використовують файлоподібний інтерфейс, але, на відміну від перших, придатні також для обміну даними мережею.

Сокет – це абстрактний об'єкт для позначення одного з кінців мережевого з'єднання, в тому числі і через Internet. Сокети Windows бувають двох типів: сокети дейтаграм і сокети потоків. Інтерфейс Windows Sockets (WinSock) заснований на BSD-версії сокетів, але в ньому є також розширення, специфічні для Windows.

Повідомлення в Windows (віконні повідомлення). Говорячи про Windows не можна не згадати про такі поняття як windows (вікна), messages (повідомлення), message queue (черга повідомлень) тощо.

Window – це прямокутна область екрану, в якій додаток відображає інформацію і отримує інформацію від користувача. Вікна належать потокам. Потік, який створив вікно вважається власником цього вікна. Потік може бути власником кількох вікон.

Вікна управляються повідомленнями. Всі події, що відбуваються з вікном, супроводжуються посилкою йому повідомлень: створення та знищення вікна, введення з клавіатури, переміщення миші, перемалювання і переміщення вікна тощо. Повідомлення вікну можуть надсилатися як самою системою, так і

додатками користувача. Кожному вікну приписана функція, яка називається віконною процедурою (window procedure), і яка викликається при обробці повідомлення.

Повідомлення можна надсилати не тільки вікнам, а й самому потоку. Кожен потік, який володіє вікном, має чергу повідомлень. Як правило, потік, що володіє вікнами, тільки тим і займається, що обробляє повідомлення, що посилаються його вікнам.

Контрольні питання і тести до розділу 12

Контрольні питання

1. Які види (рівні) планування використовуються в сучасних ОС?
2. Для чого призначене довгострокове планування?
3. Які основні задачі середньострокового планування?
4. Чому короткострокове планування (диспетчеризація або планування процесора) є найважливішим видом планування?
5. Відрізняють дві основні стратегії планування багатозадачності. Що це за стратегії?
6. У яких сучасних ОС, орієнтованих на високопродуктивне виконання додатків, реалізовані витісняючи алгоритми планування потоків (процесів)?
7. Назвіть ОС, в яких ефективно використовуються алгоритми невитісняючого планування.
8. Яка концепція лежить в основі багатьох витісняючих алгоритмів планування?
9. Як відповідно до алгоритмів, заснованих на квантуванні, відбувається зміна активного потоку?
10. Які привілеї в якості компенсації за невикористані повністю кванти часу отримують потоки при подальшому обслуговуванні?
11. Які витрати на допоміжні дії залежать від величини кванта часу?
12. Яка ще інша важлива концепція, крім квантування, лежить в основі багатьох витісняючих алгоритмів планування?
13. У багатьох ОС передбачається можливість зміни пріоритетів протягом життя потоку. Коли і ким може змінюватися пріоритет потоку?
14. Дайте визначення відносним і абсолютним пріоритетам.
15. Охарактеризуйте роботу найпростішої стратегії планування «першим прийшов, першим обслужений».
16. Як обчислюється середній час очікування і середній повний час виконання процесів?
17. Охарактеризуйте роботу найпростішої стратегії, яка заснована на ідеї кругового (карусельного) планування (round robin – RR).
18. Що може статися з алгоритмом карусельного планування при дуже великих величинах кванту часу?
19. Який основний ризик при використанні стратегії планування SPN (shortest process next – вибір найкоротшого процесу)?

20. У чому відмінність стратегії найменшого часу, що залишається (shortest remaining time – SRT), в порівнянні зі стратегією SPN?
21. Брінч Хансен розробив стратегію планування за найбільшим відносним часом реакції (highest response ratio next – HRRN). За якою формулою обчислюється динамічні пріоритети процесів при дисципліні обслуговування HRRN?
22. Чому краще використовувати дисципліну планування SPN (найкоротший процес першим), ніж FIFO, коли основною метою системи є забезпечення високої пропускної здатності?
23. Чи правда, що при використанні принципу планування HRRN, виконання коротких процесів завжди буде призначатися раніше, ніж довших? (Так, Ні)
24. Планувальник процесів визначає область пам'яті, в яку буде поміщений новий процес? (Так, Ні)
25. Які характеристики властиві процесам ОС Windows 2000 (W2K)?
26. За допомогою яких механізмів процеси користувача одержують доступ до функцій ядра ОС Windows 2000 (W2K)?
27. Які атрибути властиві процесам в ОС Windows 2000 (W2K)?
28. У яких станах може бути потік в ОС W2K в ході свого існування?
29. Які механізми синхронізації потоків використовуються в ОС W2K?
30. Які механізми взаємодії між процесами використовуються в ОС Windows 2000?

Тести

1. Основна відмінність між довгостроковим і короткостроковим плануванням (диспетчеризацією) полягає в:
    1) частоті виконання;
    2) черговості виконання;
    3) швидкості виконання;
    4) тривалості виконання.
2. Пріоритет, який змінюється під час виконання процесу, називається . . . пріоритетом.
    1) фіксованим;
    2) статичним;
    3) циклічним;
    4) динамічним.
3. Що таке планування завантаження процесора (CPU scheduling)?
    1) вибір чергового завдання для запуску і виділення йому кванта часу;
    2) аналіз статистики використання процесора;
    3) складання плану використання процесора;
    4) переривання процесора для виконання введення-виведення.
4. До якого виду планування процесів належить планування завдань?
    1) середньострокове планування;
    2) короткострокове планування;
    3) довгострокове планування;

4) довгострокове і середньострокове планування.

5. До якого виду планування належить процедура свопинга (swapping)?
   1) короткострокове планування;
   2) середньострокове планування;
   3) довгострокове планування;
   4) довгострокове і середньострокове планування.
6. До якого виду планування належить планування використання процесора?
   1) короткострокове планування;
   2) середньострокове планування;
   3) довгострокове планування;
   4) довгострокове і середньострокове планування.
7. При якому алгоритмі планування процесів здійснюється вибір нового процесу для виконання з початку черги процесів, що знаходяться в стані «готовність»?
   1) пріоритетне планування;
   2) Round Robin (RR);
   3) Shortest process next (SPN);
   4) First Come, First Served (FCFS).
8. При якому алгоритмі планування процесів здійснюється циклічний вибір нового процесу для виконання з початку черги процесів, що знаходяться в стані «готовність» і його виконання протягом фіксованого кванта часу?
   1) пріоритетне планування;
   2) Round Robin (RR);
   3) Shortest Process Next (SPN);
   4) First Come, First Served (FCFS).
9. При якому алгоритмі планування процесів з черги процесів, що знаходяться в стані «готовність», вибирається процес з мінімальною тривалістю часу безперервного використання процесора (CPU burst)?
   1) пріоритетне планування;
   2) Round Robin (RR);
   3) Shortest Process Next (SPN);
   4) First Come, First Served (FCFS).
10. При якому алгоритмі планування процесів кожному процесу присвоюється певне числове значення, відповідно до якого йому виділяється процесорний час?
   1) пріоритетне планування;
   2) Round Robin (RR);
   3) Shortest Process Next (SPN);
   4) First Come, First Served (FCFS).
11. Відомо, що програма А виконується в монопольному режимі за 10 хвилин, а програма В – за 20 хвилин, тобто при послідовному виконанні вони вимагають 30 хвилин. Якщо Т – час виконання обох цих завдань в режимі мультипрограмування, то яка з наступних нерівностей справедлива:
   1) $10 < T < 30$;
   2) $20 < T < 30$;

3) 10 <T <20;
4) T> 30.

12. До якого з перерахованих алгоритмів прагне поведінка алгоритму Round Robin (RR) у міру збільшення кванта часу?
    1) Shortest Process Next (SPN);
    2) Shortest Job First (SJF);
    3) First Come, First Served (FCFS);
    4) пріоритетне планування.
13. Нехай в обчислювальну систему надійшло три процеси з різним часом виконання за такою схемою:

№ процесу	Час виконання
1	4
2	2
3	3

Чому дорівнює середній час очікування процесу при використанні невитісняючого алгоритму «вибір найкоротшого процесу» (shortest process next – SPN)?
1) 2.3;
2) 3.3;
3) 2.6;
4) 3.6.

14. Нехай в обчислювальну систему надходять три процеси з різним часом виконання і з різними пріоритетами (3 – вищий) за наступною схемою:

№ процесу	Час надходження в систему	Час виконання	Пріоритет
1	3	4	3
2	2	2	2
3	0	3	1

Чому дорівнює середній час очікування процесу між стартом процесу і його завершенням при використанні витісняючого пріоритетного планування?
1) 3;
2) 4;
3) 5;
4) 6.

15. Нехай в обчислювальну систему надходять три процеса з різним часом виконання за такою схемою:

№ процесу	Час надходження в систему	Час виконання
1	1	1
2	1	2
3	0	4

Чому дорівнює середній час очікування процесу між стартом процесу і його завершенням при використанні витісняючої стратегії найменшого залишку часу (shortest remaining time – SRT)?

1) 3;
2) 2;
3) 1;
4) 4.

16. Нехай в обчислювальну систему надійшло три процеси з різним часом виконання за такою схемою:

№ процесу	Час виконання
1	4
2	2
3	3

Чому дорівнює середній повний час виконання процесу при використанні невитісняючого алгоритму «вибір найкоротшого процесу» (shortest process next – SPN)?

1) 4.3;
2) 4.6;
3) 5.3;
4) 5.6.

17. Нехай в обчислювальну систему надходять три процеси з різним часом виконання і з різними пріоритетами (3 – вищий) за такгою схемою:

№ процесу	Час надходження в систему	Час виконання	Пріоритет
1	3	4	3
2	2	2	2
3	0	3	1

Чому дорівнює середній повний час виконання процесу між стартом процесу і його завершенням при використанні витісняючого пріоритетного планування?

1) 4;
2) 5;
3) 7;
4) 8.

18. Нехай в обчислювальну систему надходять три процеси з різним часом виконання за такою схемою:

№ процесу	Час надходження в систему	Час виконання
1	1	1
2	1	2
3	0	4

Чому дорівнює середній час очікування процесу між стартом процесу і його завершенням при використанні витісняючої стратегії найменшого залишку часу (shortest remaining time – SRT)?

1) 3.3;
2) 4.3;
3) 3.6;
4) 4.6.

19. Нехай в обчислювальну систему надходять три процеси з різним часом виконання за такою схемою:

№ процесу	**Час виконання**
1	3
2	2
3	1

Чому дорівнює середній час очікування процесу між стартом процесу і його завершенням при використанні витісняючої стратегії кругового (карусельного) планування (round robin – RR) з квантом часу 1?

1) 2.3;
2) 2.6;
3) 3.3;
4) 3.6.

20. Нехай в обчислювальну систему надходять три процесу з різним часом виконання за такою схемою:

№ процесу	**Час виконання**
1	3
2	2
3	1

Чому дорівнює середній повний час виконання процесу між стартом процесу і його завершенням при використанні витісняючої стратегії кругового (карусельного) планування (round robin – RR) з квантом часу 1?

1) 3.3;
2) 3.6;
3) 4.3;
4) 4.6.

21. Який з перерахованих алгоритмів допускає необмежено довге відкладання вибірки одного з готових процесів на виконання?

1) пріоритетне планування;
2) Round Robin (RR);
3) Shortest Process Next (SPN);
4) First Come, First Served (FCFS).

22. Виберіть вірне твердження:

1) в системах з абсолютними пріоритетами виконання активного потоку переривається, якщо в черзі готових потоків з'явився потік, що має більший пріоритет;

2) в системах з абсолютними пріоритетами виконання активного потоку триває до тих пір, поки він сам не покине процесор.

23. Виберіть вірне твердження:
    1) в системах з відносними пріоритетами виконання активного потоку переривається, якщо в черзі готових потоків з'явився потік, що має більший пріоритет;
    2) в системах з відносними пріоритетами виконання активного потоку триває до тих пір, поки він сам не покине процесор.
24. Якщо квант часу стане більше, сумарні накладні витрати на перемикання потоку будуть:
    1) не менше;
    2) менше;
    3) не більше;
    4) більше.
25. Багатозадачність на основі режиму розподілу часу називається:
    1) кооперативною;
    2) невитісняючою;
    3) спільною;
    4) витісняючою.
26. Невитісняючі алгоритми планування засновані на такій концепції:
    1) рішення про переключення процесора з одного потоку на інший приймає ОС;
    2) активний потік виконується до тих пір, поки він сам не віддасть управління ОС;
    3) рішення про переключення процесора з одного потоку на інший приймає активний потік.
27. При появі в системі більш пріоритетного готового до виконання потоку при обслуговуванні з відносними пріоритетами виконання поточного потоку:
    1) не переривається;
    2) часто переривається;
    3) зупиняється.
28. Змінна величина кванта на спадання вигідна:
    1) коротким задачам;
    2) всім задачам;
    3) довгим задачам;
    4) користувачам.
29. Змінна величина кванта на зростання вигідна:
    1) користувачам;
    2) коротким задачам;
    3) довгим задачам;
    4) всім задачам.
30. Процес P1 оцінює необхідний йому час обслуговування в 3 секунди, а процес P2 – в 6 секунд. При цьому процес P1 очікує вже протягом 9 секунд, а процес P2 – протягом 12 секунд. Який процес буде запущений першим

відповідно до дисципліни HRRN (планування процесора за найбільшим відносним часом реакції)?

1) P1;
2) P2;
3) P1 і P2 одночасно.

31. Планувальник якого рівня повинен залишатися резидентним в основний пам'яті?
    1) довгострокового планування (планування верхнього рівня);
    2) середньострокового планування (планування проміжного рівня);
    3) планування вищого рівня;
    4) короткострокового планування (планування нижнього рівня).
32. У функції якого планувальника з перерахованих нижче входить здійснення свопинга:
    1) середньостроковий планувальник;
    2) довгостроковий планувальник;
    3) планувальник введення-виведення;
    4) короткостроковий планувальник.
33. Які базові концепції застосовуються алгоритмом планування потоків в ОС W2K?
    1) квантування і пріоритети;
    2) квантування;
    3) пріоритети.
34. В ОС Windows 2000 при виникненні переривання таймера, процедура його обробки віднімає з кванта потоку величину рівну:
    1) 1;
    2) 2;
    3) 3;
    4) 4.
35. В ОС Windows 2000 потік переходить у стан виконання із стану:
    1) очікування;
    2) транзит;
    3) готовий;
    4) резервний.
36. Для того щоб забезпечити хороший час реакції системи, алгоритм планування ОС Windows 2000 використовує разом з квантуванням концепцію:
    1) абсолютних пріоритетів;
    2) відносних пріоритетів;
    3) абсолютних і відносних пріоритетів.

13 БАГАТОПРОЦЕСОРНІ СИСТЕМИ

Поняття про процесор як про найцінніший ресурс обчислювальної системи, звичайне для перших двох десятиліть розвитку сучасної обчислювальної техніки, зараз вже практично застаріло. Зараз з появою мультипроцесорних архітектур набагато більшого значення набувають проблеми надійності, паралелізму в обчисленнях, оптимальних схем комутації та змагань між процесорами, що намагаються отримати доступ до одних і тих же ресурсів.

З перших днів свого існування комп'ютерна промисловість постійно прагнула до досягнення все більшої і більшої обчислювальної потужності. Наприклад, комп'ютер ENIAC міг виконувати 300 операцій в секунду, з легкістю тисячократно обставляючи будь-який попередній йому калькулятор, але людей і це не влаштовувало. Швидкодія сучасних машин в мільйони разів перевищує можливості ENIAC, але є потреби в ще більшій потужності.

З моменту появи фактично діючих електронно-обчислювальних машин вчені і практики в галузі інформатики працювали над тим, щоб ці машини можна було об'єднувати в так звані багатомашинні комплекси. Справедливо передбачалося, що такі комплекси будуть набагато продуктивніше і ефективніше, ніж кожна окрема обчислювальна машина. При цьому розглядалися різні можливі варіанти об'єднання комп'ютерів між собою.

13.1 Історія багатопроцесорної обробки

Для розв'язання великих завдань потрібні все більш швидкі комп'ютери. Є всього два основних способи підвищення швидкодії ЕОМ [38]:

1. За рахунок підвищення швидкодії елементної бази (тактової частоти). Швидкодія процесора зростає пропорційно зростанню тактової частоти, при цьому не потрібно зміни системи програмування і програм користувача.
2. За рахунок збільшення числа одночасно працюючих в одному ЕОМ для розв'язання одного завдання, процесорів тощо, тобто за рахунок паралелізму виконання операцій. Це вимагає використання складних систем паралельного програмування.

Паралельні системи по архітектурі поділяються на два класи:

1. Конвеєрні системи, коли кілька спеціалізованих блоків одночасно працюють над частинами одного потоку команд.
2. Паралельні системи, коли певна кількість команд однієї програми одночасно виконуються множиною АЛУ або процесорів.

Тактова частота. У минулому проблема обчислювальної потужності завжди вирішувалася за рахунок підвищення тактової частоти. На жаль, цьому підвищенню вже починає перешкоджати ряд фундаментальних обмежень. Так як електричний сигнал не може поширюватися швидше за швидкість світла, яка дорівнює у вакуумі приблизно 30 см/нс, а в мідному провіднику або в оптичному кабелі швидкість поширення сигналу дорівнює приблизно 20 см/нс. З цього випливає, що на комп'ютері з тактовою частотою 10 ГГц сигнал не може

подолати за один такт сумарну відстань, що перевищує 2 см. Для комп'ютера з тактовою частотою 100 ГГц максимальна сумарна довжина шляху дорівнює 2 мм. Комп'ютер з тактовою частотою 1 ТГц (1000 ГГц) повинен бути менше 100 мкм (0,1 мм), щоб сигнал міг дістатися з одного його кінця до іншого за один такт [9].

Зменшити комп'ютери до таких розмірів, може бути, і можливо, але тоді на заваді стане інша фундаментальна проблема: відвід тепла. Чим швидше комп'ютер, тим більше тепла він виділяє, а чим він менший, тим важче від цього тепла позбутися. Вже зараз на потужних x86-системах системи охолодження, встановлені на процесорі, більше самого процесора.

Конвеєрні системи. Історично ідея конвеєрної обробки стосовно до обчислювальної техніки першою отримала теоретичне обґрунтування і практичне втілення в реальній апаратурі. Конвеєр – найбільш «дешевий» спосіб підвищення продуктивності за рахунок введення паралелізму.

Ідея конвеєрної обробки полягає у виділенні окремих етапів виконання спільної операції, причому кожен етап, виконавши свою роботу, передавав би результат наступного, одночасно приймаючи нову порцію вхідних даних. Отримуємо очевидний виграш в швидкості обробки за рахунок поєднання раніше рознесених в часі операцій.

Припустимо, що в операції можна виділити п'ять мікрооперацій, кожна з яких виконується за одну одиницю часу. Якщо кожну мікрооперацію виділити в окремий етап (або інакше кажуть – ступінь) конвеєрного пристрою, то на п'ятій одиниці часу на різній стадії обробки такого пристрою будуть знаходитися перші п'ять пар аргументів, а весь набір зі ста пар буде оброблений за 5+99 = 104 одиниці часу. В ідеальному випадку прискорення в порівнянні з послідовним пристроєм зростає в п'ять разів (по числу ступенів конвеєра) [38]. Конвеєрні системи втрачають сенс, коли час передачі інформації з рівня на рівень стає порівнянним з часом обчислень на кожному ступені.

Паралельні системи. З моменту появи фактично діючих електронно-обчислювальних машин вчені і практики в галузі інформатики працювали над тим, щоб ці машини можна було об'єднувати в так звані багатомашинні комплекси. Передбачалося, що такі комплекси будуть набагато продуктивніше і ефективніше, ніж кожна окрема обчислювальна машина, а головне – набагато відмовостійкими. Очевидно, що виконувати вимоги відмовостійкості обчислювальній системі при єдиному процесорі неможливо, тому всі відмовостійкі системи, незалежно від реалізацій і архітектурних рішень, є багатопроцесорними.

Один з підходів до збільшення швидкості полягає в широкомасштабному застосуванні паралельних обчислювальних систем. Ці системи містять багато центральних процесорів, кожний з яких працює на звичайній частоті (яке б значення вона не мала в даний час), але в порівнянні з окремо взятим процесором всі разом вони мають куди більш високу обчислювальну потужність. Зараз вже продаються системи, що складаються з десятків тисяч центральних процесорів. А в лабораторіях вже створені системи з 1 млн центральних процесорів [9].

Неважко буде поставити в одній дуже великій кімнаті тисячу не пов'язаних між собою комп'ютерів за умови, що вистачить на це коштів. Розмістити тисячі комп'ютерів по всьому світу ще легше, оскільки при цьому не потрібно шукати для них відповідну велику кімнату. Проблеми починаються, коли потрібно організувати обмін даними між комп'ютерами для спільної роботи при розв'язанні єдиного завдання. Тому був пророблений великий обсяг роботи по розробці технології з'єднання комп'ютерів, а різні технології з'єднання привели до якісних систем, які відрізняються одна від одної типами систем і різним організаціям програмного забезпечення.

Весь обмін даними між електронними компонентами в кінцевому підсумку зводиться до обміну повідомленнями – чітко визначеними бітовими рядками. Різниця полягає у використовуваних масштабах часу, відстані і логічної організації. На одному полюсі знаходиться багатопроцесорна система із загальним простором пам'яті, де від двох до тисячі центральних процесорів обмінюються даними через загальну пам'ять. У цій моделі кожен центральний процесор має рівний доступ до всієї фізичної пам'яті і може читати, і записувати окремі слова. Доступ до слова пам'яті зазвичай займає 5-50 нс. Зараз вже немає нічого незвичайного в розміщенні на один кристал центрального процесора більш одного обчислювального ядра з наданням ядрам спільного доступу до основної пам'яті (а іноді навіть і до спільним блокам кеш-пам'яті).

Багатопроцесорна обробка зародилася в середині 1950-х в ряді компаній (IBM, Control Data Corporation). На початку 1960-х Burroughs Corporation представила симетричний мультипроцесор типу MIMD з чотирма CPU, що має до шістнадцяти модулів пам'яті, з'єднаних координатним з'єднувачем (перша архітектура SMP, Symmetric MultiProcessing – технологія симетричної мультипроцесорності) [9]. Широко відомий і успішний комп'ютер CDC 6600 був представлений в 1964 році і забезпечував CPU десятьма підпроцесорами (периферійними процесорами). В кінці 1960-х Honeywell випустила іншу симетричну мультипроцесорну систему з восьми CPU Multics.

У той час як багатопроцесорні системи розвивалися, інші технології також йшли вперед, зменшуючи розміри процесорів і збільшуючи їх здатність працювати на значно більшій тактовій частоті. Багатопроцесорні системи, що розглядаються в цьому розділі, широко використовуються для вирішення багатьох задач в науці, промисловості, а також інших областях людської діяльності.

Ще одна область розвитку, що має відношення до досліджуваного питання, – це неймовірно бурхливе зростання мережі Інтернет. Система, що складається з тисячі комп'ютерів, розосереджених по всьому світу, не відрізняється від системи, що складається з тисячі комп'ютерів, що знаходяться в одному приміщенні, хоча затримки по часу і інші технічні характеристики у цих двох систем розрізняються. Ці системи також будуть коротко розглянуті в цьому розділі.

13.2 Форми паралелізму

Паралелізм – це можливість одночасного виконання більш однієї арифметико-логічного операції або програмної гілки. Вивчення ряду алгоритмів і програм показало, що можна виділити такі основні форми паралелізму [38]:

– дрібнозернистий паралелізм;
– крупнозернистий паралелізм.

Дрібнозернистий паралелізм (паралелізм суміжних операцій або скалярний паралелізм) забезпечується за рахунок паралелізму всередині базових блоків (5-20 команд), які є частинами програм, що не містять умовних і безумовних переходів. Цей вид паралелізму реалізується блоками одного процесора (різними АЛП, помножувачами, блоками звернення до пам'яті, зберігання адреси, переходів тощо) і навіть при оптимальному плануванні паралелізм не може бути більшим.

Крупнозернистий паралелізм забезпечується за рахунок паралелізму незалежних програмних гілок, підпрограм, потоків (ниток) всередині програм і реалізується процесорами або ядрами багатоядерних процесорів. Крупнозернистий паралелізм включає векторний паралелізм і паралелізм незалежних гілок.

Векторний паралелізм. Найбільш поширеною в обробці структур даних є векторна операція (природний паралелізм).

Паралелізм незалежних гілок. Суть паралелізму незалежних гілок полягає в тому, що в програмі розв'язання великого завдання можуть бути виділені програмні частини, незалежні за даними.

Ефективність паралельних обчислень можна обчислити за допомогою закону Амдала. Джин Амдал розробляв в IBM комп'ютерні архітектури, але популярність йому приніс його закон, в якому розраховується максимально можливе поліпшення (прискорення – R) системи при поліпшенні її частини. Закон використовується для обчислення максимального теоретичного поліпшення роботи системи при використанні декількох процесорів:

$$\textit{Прискорення } (R) = 1/(F + (1-F)/N).$$

Використовуючи це рівняння, можна обчислити максимальне поліпшення продуктивності системи, що використовує N процесорів і фактор F, який показує, яка частина системи не може бути розпаралелена (частина системи, яка є послідовною за своєю природою).

Через те, що не всі в задачі можуть бути розпаралелені і є непродуктивні витрати в управлінні процесорами, прискорення виявляється трохи менше.

Нехай, наприклад, $F = 0{,}2$ (що є реальним значенням), тоді прискорення не може перевищувати 5 при будь-якому числі процесорів, тобто максимальне прискорення визначається потенційним паралелізмом завдання.

Якщо система має кілька архітектурних рівнів з різними формами паралелізму, то якісно загальне прискорення в системі буде:

$$R = r1 \times r2 \times r3,$$

де r_i – прискорення деякого рівня.

13.3 Класифікація обчислювальних систем

Під обчислювальною системою розуміють сукупність взаємопов'язаних і взаємодіючих процесорів або ЕОМ, периферійного обладнання та програмного забезпечення, призначену для збору, зберігання, обробки і розподілу інформації.

Термін «***обчислювальна система***» з'явився в середині 60-х років з появою ЕОМ третього покоління. Його часто використовують стосовно однопроцесорних комп'ютерів. Однак загальним тут є підкреслення можливості побудови паралельних гілок в обчисленнях, що не передбачалося класичною структурою ЕОМ.

Відмінною особливістю обчислювальних систем по відношенню до ЕОМ є наявність в них декількох обчислювачів, що реалізують паралельну обробку. Паралелізм – основа високопродуктивної роботи всіх підсистем обчислювальних машин. Організація пам'яті будь-якого рівня ієрархії, організація системного введення-виведення, організація мультиплексування шин базуються на принципах паралельної обробки запитів. Сучасні операційні системи є багатозадачними і розрахованими на багато користувачів, імітуючи паралельне виконання програм за допомогою механізму переривань.

Створення обчислювальних систем переслідує наступні основні цілі: підвищення продуктивності системи за рахунок прискорення процесів обробки даних, підвищення надійності та достовірності обчислень, надання користувачам додаткових сервісних послуг тощо [37].

Існує велика кількість ознак, за якими класифікують обчислювальні системи: за цільовим призначенням і виконуваних функцій, за типами і кількістю ЕОМ або процесорів, з архітектури системи та інші. Проте основними з них є ознаки структурної і функціональної організації обчислювальної системи.

За призначенням обчислювальні системи ділять на універсальні і спеціалізовані. Універсальні обчислювальні системи призначаються для розв'язання найрізноманітніших задач. Спеціалізовані системи орієнтовані на розв'язання вузького класу задач.

За типом обчислювальні системи поділяються на багатомашинні і багатопроцесорні обчислювальні системи. Багатомашинні обчислювальні системи (БМС) з'явилися історично першими. Основні відмінності БМС полягають, як правило, в організації зв'язку та обмін інформацією між ЕОМ комплексу. Кожна з них зберігає можливість автономної роботи і управляється власною ОС. Будь-яка інша підключена ЕОМ комплексу розглядається як периферійне спеціальне обладнання. Залежно від територіальної роз'єднаності ЕОМ і використовуваних засобів сполучення забезпечується різна оперативність їх інформаційної взаємодії.

Багатопроцесорні системи (БПС) будуються при об'єднанні кількох процесорів. В якості єдиного ресурсу вони мають загальну оперативну пам'ять (ЗОП). Паралельна робота процесорів і використання ЗОП забезпечується під управлінням єдиної операційної системи. Багато дослідників вважають, що використання БПС є основним магістральним шляхом розвитку обчислювальної техніки нових поколінь [25; 28].

Однак БПС мають і суттєві недоліки. Вони, в першу чергу, пов’язані з ресурсами ЗОП. При великій кількості процесорів комплексу можливе виникнення конфліктних ситуацій, коли кілька процесорів звертаються з операціями типу «читання» і «запис» до одних і тих же областей пам’яті.

Крім процесорів до ЗОП також підключаються всі канали (процесори введення-виведення), засоби вимірювання часу тощо. Тому другим серйозним недоліком БПС є проблема комутації абонентів і доступу їх до ЗОП. Від того, наскільки вдало вирішуються ці проблеми, і залежить ефективність застосування БПС. Це рішення повинно забезпечуватися апаратно-програмними засобами. Процедури взаємодії дуже ускладнюють структуру ОС БПС. Накопичений досвід побудови подібних систем показав, що вони ефективні при невеликому числі процесорів.

За типом ЕОМ або процесорів, які використовуються для побудови обчислювальних систем, розрізняють *однорідні* і *неоднорідні системи*. ***Однорідні системи*** припускають об’єднання однотипних ЕОМ (процесорів), ***неоднорідні*** – різнотипних. В однорідних системах значно спрощується розробка і обслуговування технічних і програмних (в основному ОС) коштів. У них забезпечується можливість стандартизації та уніфікації з’єднань і процедур взаємодії елементів системи.

За ступенем територіальної роз’єднаності обчислювальних модулів обчислювальні системи діляться на *системи суміщеного* (*зосередженого*) і *розподіленого* (*роз’єднаного*) типів. Зазвичай такий розподіл стосується лише БМС. Багатопроцесорні системи відносяться до систем суміщеного типу. Більш того, з огляду на успіхи мікроелектроніки, це поєднання може бути дуже глибоким.

За методами управління елементами обчислювальні системи розрізняють *централізовані*, *децентралізовані* і *зі змішаним керуванням*. Крім паралельних обчислень, вироблених елементами системи, необхідно виділяти ресурси на забезпечення управління цими обчисленнями. У централізованих обчислювальних системи за це відповідає головна, або диспетчерська ЕОМ (процесор). Її завданням є розподіл навантаження між елементами, виділення ресурсів, контроль стану ресурсів, координація взаємодії. Централізований орган управління в системі може бути жорстко фіксований або ці функції можуть передаватися іншій ЕОМ (процесору), що сприяє підвищенню надійності системи. Централізовані системи мають більш прості ОС.

У децентралізованих системах функції управління розподілені між її елементами. Кожна ЕОМ (процесор) системи зберігає відому автономію, а необхідна взаємодія між елементами встановлюється за спеціальними наборами сигналів. З розвитком обчислювальних систем і, зокрема, мереж ЕОМ, інтерес до децентралізованих систем стає дедалі більше.

У системах зі змішаним керуванням поєднуються процедури централізованого та децентралізованого управління. Перерозподіл функцій здійснюється в ході обчислювального процесу, виходячи з ситуації, що склалася.

За архітектурою системи. Оскільки обчислювальні системи з’явилися як паралельні системи, то розглянемо ще раз класифікацію архітектур,

запропоновану М. Флінном на початку 60-х років, с цієї точки зору (рис. 13.1). В її основу закладено два можливих види паралелізму: незалежність потоків завдань (команд), які існують в системі, і незалежність даних, які обробляються в кожному потоці. Згідно з цією класифікацією існує чотири основних архітектури обчислювальних систем.

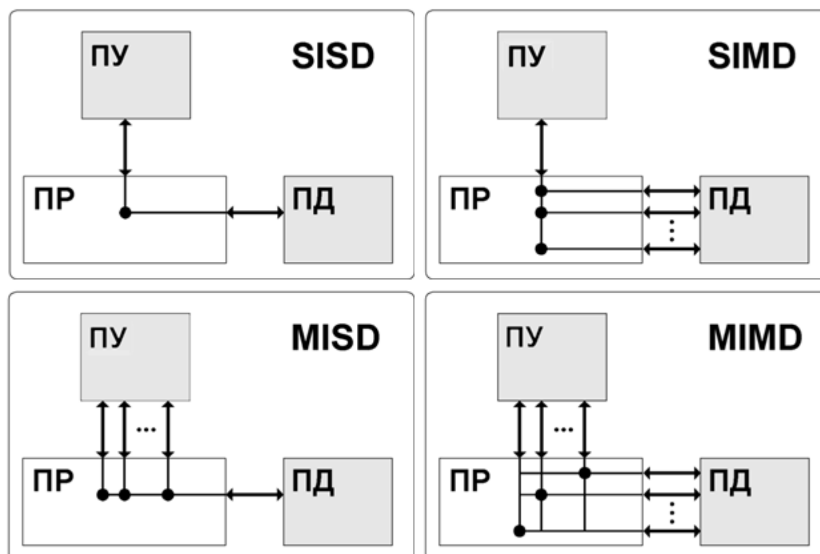


Рисунок 13.1 – Класифікація архітектур обчислювальних систем Флінна

Архітектура ОКОД (одиночний потік команд – одиночний потік даних, SISD) охоплює всі однопроцесорні і одномашинні варіанти систем, тобто системи з одним обчислювачем. Всі ЕОМ класичної структури потрапляють в цей клас. Тут паралелізм обчислень забезпечується шляхом поєднання виконання операцій окремими блоками АЛП, а також паралельною роботою пристроїв введення-виведення інформації та процесора. Закономірності організації обчислювального процесу в цих структурах досить добре вивчені.

Архітектура ОКМД (одиночний потік команд – множинний потік даних, SIMD) передбачає створення структур векторної або матричної обробки. Системи цього типу зазвичай будуються як однорідні. Процесорні елементи, що входять в систему, ідентичні, і всі вони підкоряються одній і тій же послідовності команд. Однак кожен процесор обробляє свій потік даних.

Під цю схему добре підходять задачі обробки матриць або векторів (масивів), задачі розв'язання систем лінійних і нелінійних, алгебраїчних і диференціальних рівнянь. У структурах даної архітектури бажано забезпечувати з'єднання між процесорами, які б нагадували матрицю, де кожен процесорний елемент пов'язаний з сусідніми. Векторний або матричний тип обчислень є необхідним атрибутом будь-якій супер-ЕОМ.

Архітектура МКОД (множинний потік команд – одиночний потік даних, MISD) передбачає побудову своєрідного процесорного конвеєра, в якому результати обробки передаються від одного процесора до іншого по ланцюжку. Вигоди такого виду обробки зрозумілі. Однак у більшості алгоритмів дуже важко виявити подібний, регулярний характер в обчисленнях. Крім того, на практиці не можна забезпечити і «велику довжину» такого конвеєра, при якій досягається

найвищий ефект. Разом з тим конвеєрна схема знайшла застосування в так званих скалярних процесорах супер-ЕОМ, в яких вони застосовуються як спеціальні процесори для підтримки векторної обробки.

Архітектура МКМД (множинний потік команд – множинний потік даних, MIMD) передбачає, що всі процесори системи працюють з різними програмами і з індивідуальним набором даних. У найпростішому випадку вони можуть бути автономні і незалежні. Така схема використання обчислювальних систем часто застосовується на багатьох великих обчислювальних центрах для збільшення пропускної спроможності центру. Такі системи зазвичай називають багатопроцесорними. Цей клас поділяється на два великих підкласи:

1. З загальною для всіх процесорів пам’яттю. Це спрощує обладнання, але кількість процесорів обмежена пропускною здатністю пам’яті.
2. З індивідуальною пам’яттю для кожного процесора. Число процесорів не обмежується, але міжпроцесорний обмін даними знижує потенційне прискорення.

З плином часу з’явилася різновид цієї технології, яка називається SPMD (Single Program – Multiple Data), яка використовується в багатопроцесорних системах з загальною пам’яттю SMP (Symmetric Multiprocessing).

Побудова обчислювальної системи будь-якого типу передбачає, що модулі, що об’єднуються в систему, повинні бути сумісні. Поняття сумісності включає три аспекти: апаратурний, або технічний, програмний та інформаційний. Технічна (HardWare) сумісність передбачає виконання наступних умов:

– підключена один до одного апаратура повинна мати єдині стандартні, уніфіковані засоби з’єднання: кабелі, число проводів в них, єдине призначення проводів, роз’єми, заглушки, адаптери, плати тощо;
– параметри електричних сигналів, якими обмінюються технічні пристрої, повинні відповідати один одному: амплітуди імпульсів, полярність, тривалість тощо;
– алгоритми взаємодії (послідовності сигналів по окремим проводам) не повинні вступати в протиріччя один з одним.

Останній пункт тісно пов’язаний з програмною сумісністю. Програмна сумісність (SoftWare) вимагає, щоб програми, що передаються з одного технічного засобу в інший (між ЕОМ, процесорами, між процесорами і зовнішніми пристроями) були правильно зрозумілі і виконані іншим пристроєм.

Якщо обмінюються пристрої ідентичні один одному, то проблем зазвичай не виникає. Якщо взаємодіючі пристрої відносяться до одного і того ж сімейства ЕОМ, але стикуються різні моделі, то сумісність забезпечується «знизу-вгору», тобто раніше створені програми повинні виконуватися і на новітніх моделях, але не навпаки. Якщо ж стикується апаратура, яка має абсолютно різну систему команд, то слід обмінюватися вихідними модулями програм з подальшою їх трансляцією.

Інформаційна сумісність передбачає, що інформаційні масиви, які передаються, будуть однаково інтерпретуватися модулями обчислювальної

системи. Повинні бути стандартизовані алфавіти, розрядність, формати, структура і розмітка файлів тощо.

Серед розглянутих структур обчислювальних систем МКМД-структури є найбільш цікавим класом структур обчислювальних систем для досягнення найкращого паралелізму. Уже з назви МКМД-структур видно, що в даних системах можна знайти всі перераховані види паралелізму. Цей клас дає велику різноманітність структур, які сильно відрізняються один від одного своїми характеристиками (рис. 13.2).

Важливу роль в обчислювальних системах відіграють способи взаємодії комп'ютерів або процесорів у системі. У сильнозв'язаних системах досягається висока оперативність взаємодії процесорів за допомогою загальної оперативної пам'яті. При цьому користувач має справу з багатопроцесорною організацією.

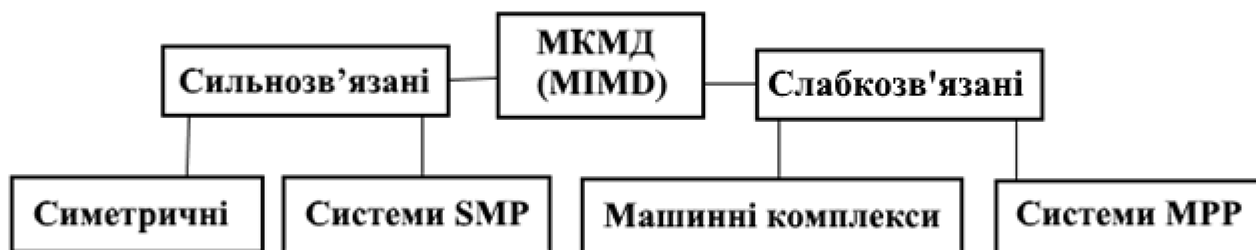


Рисунок 13.2 – Типові структури обчислювальних систем в МКМД-класі

Найбільш простими за будовою і організацією функціонування є ***однорідні***, ***симетричні структури***. Вони забезпечують простоту підключення процесорів і не вимагають дуже складних централізованих операційних систем, що розміщуються на одному з процесорів.

Однак при побудові таких систем виникає багато проблем з використанням загальної оперативної пам'яті. Число об'єднаних процесорів не може бути велике, воно не перевищує 16. Для зменшення числа звернень до пам'яті і конфліктних ситуацій може використовуватися багатоблокова побудова ОП, функціональне закріплення окремих блоків за процесорами, постачання процесорів з власною пам'яттю типу кеш. Але всі ці методи не вирішують проблеми підвищення продуктивності обчислювальних систем в цілому. Апаратурні витрати при цьому істотно зростають, а продуктивність систем збільшується незначно.

Поява потужних мікропроцесорів типу Pentium призвело до експериментів зі створення багатопроцесорних систем на їх основі. Так, для включення потужних серверів в локальні мережі персональних комп'ютерів була запропонована дещо змінена структура використання ЗОП – SMP (Shared Memory multiProcessing – ***мультипроцесування з розподілом пам'яті***). На загальній шині оперативної пам'яті можна об'єднати кілька мікропроцесорів.

Слабкозв'язані МКМД-системи можуть будуватися як багатомашинні комплекси або використовувати в якості засобів передачі інформації загальне поле зовнішньої пам'яті на дискових накопичувачах великої ємності.

Невисока оперативність взаємодії заздалегідь визначає ситуації, в яких число міжпроцесорних конфліктів при зверненні до загальних даних і один до

одного було б мінімальним. Для цього необхідно, щоб комп'ютери комплексу обмінювалися один з одним з невеликою частотою, забезпечуючи автономність процесів (програми і дані до них) і паралелізм їх виконання. Тільки в цьому випадку забезпечується належний ефект. Ці проблеми вирішуються в комп'ютерних мережах.

Успіхи мікроінтегральної технології дозволяють розширити межі і цього напрямку. Можна побудувати системи з десятками, сотнями і навіть тисячами процесорних елементів, розміщуючи їх у безпосередній близькості. Якщо кожен процесор системи має власну пам'ять, то він також буде зберігати відому автономію в обчисленнях. Вважається, що саме такі системи займуть домінуюче становище у світі комп'ютерів найближчі 10-15 років. Подібні обчислювальні системи отримали назву систем з масовим паралелізмом (Mass-Parallel Processing, **MPP**).

Всі процесорні елементи в таких системах повинні бути пов'язані єдиним комутаційним середовищем. Але тут виникають проблеми, аналогічні ОКМД системам, але на новій технологічній основі.

Передача даних в MPP-системах передбачає обмін не окремими даними під централізованим управлінням, а підготовленими процесами (програмами разом з даними). Цей принцип побудови обчислень вже не відповідає принципам програмного управління класичної ЕОМ. Передача даних процесу за його готовністю більше відповідає принципам побудови «потокових машин» (машин, керованих потоками даних). Подібний підхід дозволяє будувати системи з величезною продуктивністю і реалізовувати проекти з будь-якими видами паралелізму, наприклад, перейти до «SIMD обчислень» з довільним паралелізмом. Однак, для цього необхідно вирішити цілий ряд проблем, пов'язаних з описом і програмуванням комутацій процесів і управління ними. Математична база цієї науки в даний час практично відсутня.

13.4 Варіанти об'єднання мультипроцесорних систем

Розглянемо різні можливі варіанти об'єднання комп'ютерів між собою.

Перший можливий варіант – це об'єднання між собою декількох процесорів, що працюють усередині однієї обчислювальної машини. Ця модель може бути реалізована з використанням фізично окремих центральних процесорів, декількох ядер на одному центральному процесорі або їх комбінації (мультипроцесорна модель, рис. 13.3, *а*) [9].

У цій системі кожен процесорний елемент має безпосередній доступ до пам'яті і введення-виведення. Взаємодія здійснюється через пам'ять, що розділяється [9]. Час доступу до пам'яті зазвичай становить від 10 до 50 нс.

При своїй простоті ця модель технічно реалізується не так-то просто і зазвичай включає в себе передачу великої кількості захищених повідомлень. Але програмістам довелося серйозно потрудитися, щоб зробити таке програмне забезпечення, яке в максимальній мірі використовувало б ефект від спільної роботи процесорів.

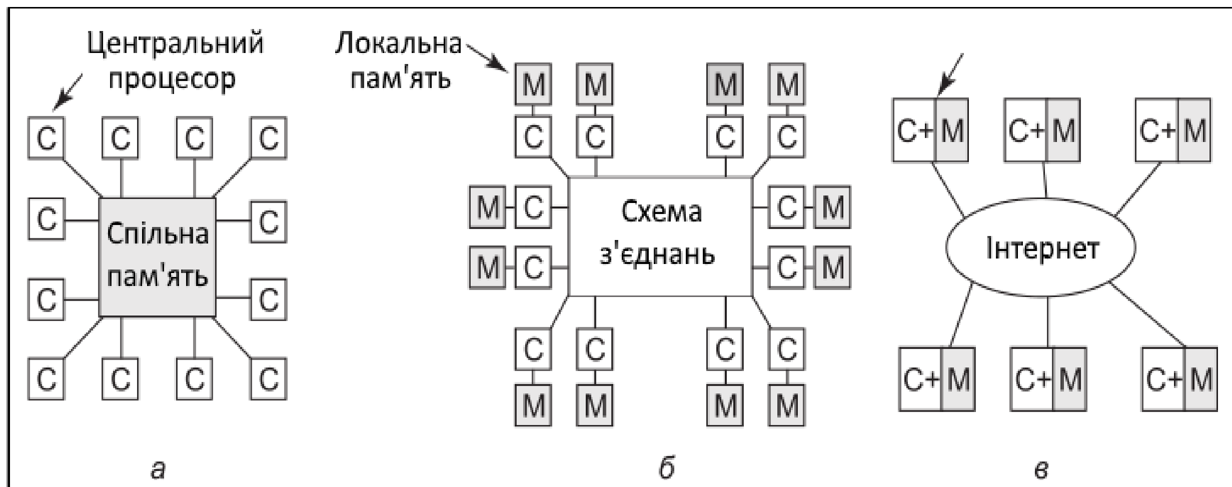


Рисунок 13.3 -Мультипроцесорна система: *а* – із загальною пам’яттю; *б* – з передачею повідомлень; *в* – глобальна розподілена система

Потрібно було домогтися того, щоб процесори працювали одночасно і, головне, паралельно. Але для цього хід виконання програми потрібно розділити на дві або більше підпрограми (в залежності від числа одночасно працюючих процесорів), з тим, щоб кожна підпрограма виконувалася за допомогою свого «власного» процесора.

Крім того, такі підпрограми повинні були в ході обробки даних обмінюватися між собою даними. Це означало, що кожен процесор може зробити запит іншому процесору, щоб отримати від нього необхідну відповідь. Але ж цей інший процесор в момент запиту може обробляти власну підпрограму або інший запит від іншого процесора, і не зможе відповісти на запит.

Така ситуація означала, що процесор, який запросив дані від другого «зайнятого» процесора, буде «простоювати» в очікуванні відповіді. А значить, швидкодія багатопроцесорного комплексу може в підсумку зійти нанівець, так як всі процесори стануть чекати, коли звільниться який-небудь єдиний і найбільш перевантажений процесор. У підсумку виходило, що швидкодія багатопроцесорного комплексу може дорівнювати швидкодії єдиного процесора, що повністю дискредитувало ідею багатопроцесорних комплексів.

Принцип зв’язку процесорів і обчислювальних машин, коли кожен процесор (або комп’ютер) зобов’язані, незважаючи ні на що, відповісти іншому процесору (або комп’ютеру) отримав найменування «***сильнозв’язані системи***».

Розробка мультипроцесорних систем у результаті привела до створення багатопроцесорних комплексів і супер-ЕОМ, що у свою чергу, дозволило вирішувати складні задачі. Але знайшлися і противники сильнозв’язаних систем, які бачили в цьому тупиковий шлях розвитку, який в перспективі не дозволяв збільшувати число пов’язаних між собою комп’ютерів понад певною кількістю.

Дійсно, сильнозв’язані системи можуть успішно працювати, якщо число компонентів такої системи не перевищує буквально кілька одиниць, може бути, десятки, але не більше того. І опоненти сильнозв’язаних систем розробили і опублікували протилежні принципи так званих «***слабкозв’язаних систем***».

1. Будь-який комп'ютер може надіслати запит будь-якому іншому комп'ютеру. Це був перший постулат, який збігався з постулатами «сильнозв'язаних систем».
2. Другий постулат говорив: комп'ютер, який отримав запит від іншого комп'ютера, має право відхилити запит.

Другий постулат слабкозв'язаних систем повністю заперечував «сильнозв'язані системи», так як дозволяв кожному комп'ютеру бути абсолютно незалежним від інших комп'ютерів.

Система, в якій декілька пар «процесор-пам'ять» з'єднані один з одним високошвидкісною схемою, називається ***мультипроцесорною системою з передачею повідомлень*** (див. рис. 13.3, *б*). Інша назва такого типу архітектури – ***кластер***. Кожен модуль пам'яті є локальним по відношенню до одного центрального процесора, і доступ до нього можна отримати тільки через цей центральний процесор. Центральні процесори зв'язуються один з одним шляхом відправки повідомлень через схему з'єднань. При наявності якісної схеми з'єднань короткі повідомлення можуть бути відправлені за 10-50 мкс. У цій конструкції не використовується загальна глобальна пам'ять. Мультипроцесорні комп'ютери (тобто системи з передачею повідомлень) набагато простіші в створенні, ніж мультипроцесори (системи із загальною пам'яттю), але вони важче в програмуванні.

Фахівці в галузі інформатики, які виросли на принципі сильнозв'язаних систем, не розуміли, яку практичну користь може принести другий постулат, який, на їхню думку, повністю руйнував зв'язок між комп'ютерами, розбивав на самостійні одиниці багатомашинний комплекс, виключаючи спільну роботу комп'ютерів.

Але прихильники слабкозв'язаних систем дивилися на це абсолютно з іншого боку, з позиції надійності і працездатності системи. Так, якщо один або декілька комп'ютерів в слабкозв'язаній системі з тих чи інших причин вийдуть з ладу, то вся система в цілому залишиться працездатною. З'явилися мережі, які були не тільки надійними, але їх ще можна було необмежено розширювати, підключаючи до них все нові і нові комп'ютери. Принципи слабкозв'язаних систем, реалізовані на практиці, забезпечили можливість об'єднання в мережі необмеженого числа комп'ютерів. Решта справи була тільки за інженерами, яким потрібно було придумати, як технічно це можна робити, особливо якщо комп'ютери перебувають на відстані один від одного.

Третя модель (див. Рис. 13.3, *в*) об'єднує повноцінні комп'ютерні системи глобальною мережею, такою як Інтернет, і утворюють разом ***розподілену систему*** (distributed system). У кожної з цих систем є власна оперативна пам'ять, і системи зв'язуються один з одним шляхом відправки повідомлень. Основна відмінність систем, показаних на рис. 13.3, *б* і *в*, полягає в тому, що в останній з них використовуються повноцінні комп'ютери, а час передачі повідомлень часто становить 10-100 мс. Настільки тривалі затримки змушують використовувати ці так звані ***слабкозв'язані розподілені системи*** дещо по-іншому, ніж

сильнозв'язані і слабкозв'язані системи (див. рис. 13.3, *а*, *б*). Час затримки у цих трьох різновидів систем різниться практично на три порядки [9].

Звертаючись до тих чи інших ресурсів розподіленої системи (Інтернет), користувачі не замислюються про те, як відбувається з'єднання з потрібним сервером. Адже на шляху кожного запиту і кожної відповіді стоїть певна кількість об'єднаних в єдину мережу комп'ютерів, кожен з яких в момент запиту або відповіді може бути зайнятий іншою справою, і не мати можливості обробити запит, може бути зламаний або, нарешті, просто вимкнений. І, тим не менш, інформація проходить по мережі, використовуючи інші канали, проходячи через інші комп'ютери та сервери. І все це відбувається завдяки реалізованому на практиці принципу слабкозв'язаних систем,

13.5 Мультипроцесор із загальною пам'яттю

Мультипроцесор із загальною пам'яттю (shared-memory multiprocessor, далі просто мультипроцесор) – комп'ютерна система, в якій два і більше центральних процесора мають повний доступ до загальної оперативної пам'яті. Програма, запущена на будь-якому з центральних процесорів, бачить звичайний віртуальний простір, що має, як правило, сторінкову організацію. Єдина незвичайна властивість, притаманна цій системі, полягає в тому, що центральний процесор може записати якесь значення в слово пам'яті, а потім зчитати це слово і отримати інше значення, тому що інший центральний процесор його вже змінив. При належній організації ця властивість формує основу для міжпроцесорного обміну даними: один центральний процесор записує якісь дані в пам'ять, а інший їх зчитує з пам'яті.

Здебільшого мультипроцесорні операційні системи мало чим відрізняються від звичайних. Вони обробляють системні виклики, здійснюють управління пам'яттю, надають файлову систему і керують пристроями введення-виведення. Проте є ряд областей, де вони мають унікальні особливості. Ці області включають синхронізацію процесів, управління ресурсами і планування. Далі ми спочатку дамо короткий огляд мультипроцесорного апаратного забезпечення, а потім перейдемо до питань, що стосуються операційних систем.

13.5.1 Мультипроцесорне апаратне забезпечення

Для простої і «дешевої» підтримки багатопроцесорної організації була запропонована архітектура SMP (Symmetric Multi-Processors) – мультипроцесування з розподілом пам'яті, що припускає об'єднання процесорів на загальній шині оперативної пам'яті (рис. 13.4). Слово «симетричний» в назві архітектури означає, що кожен процесор може робити все те, що і будь-який інший процесор. За апаратну простоту реалізації засобів SMP доводиться розплачуватися процесорним часом очікування в черзі до шини оперативної пам'яті. Коли CPU хоче прочитати слово в пам'яті, він спочатку перевіряє, чи вільна шина. Якщо шина вільна, CPU виставляє на неї адресу потрібного йому слова, подає кілька керуючих сигналів і чекає, поки пам'ять не виставить потрібне слово на шину даних.

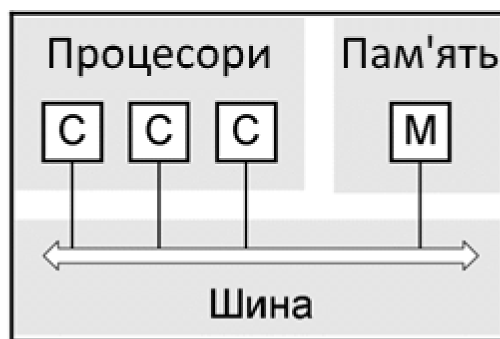


Рисунок 13.4 – Мультипроцесорна система із загальною шиною

Якщо шина зайнята, CPU просто чекає, поки вона не звільниться. У цьому полягає проблема даної архітектури. При двох або чотирьох CPU змаганням за шину можна ще управляти. Але при 32 або 64 CPU шина буде постійно зайнята, а продуктивність системи буде повністю обмежена пропускною здатністю шини. При цьому більшу частину часу CPU будуть простоювати.

У більшості випадків користувачі готові додати в сервер один або більше процесорів (але рідко – більше чотирьох) в надії збільшити продуктивність системи. Вартість цієї операції незначна в порівнянні з вартістю всього сервера, а результат найчастіше не виправдовує очікування користувача. У результаті апаратні витрати зростають, а продуктивність системи наполегливо «не бажає» збільшуватися пропорційно числу процесорів. Те, що можуть собі дозволити дорогі і складні мейнфрейми і суперкомп’ютери, не годиться для компактних багатопроцесорних серверів.

Пропускну здатність пам’яті в таких системах можна значно збільшити шляхом застосування великої багаторівневої кеш-пам’яті. При цьому кеш-пам’ять може містити як колективні, так і приватні дані.

Приватні дані **–** це дані, які використовуються одним процесором, в той час як колективні дані використовуються багатьма процесорами, по суті забезпечуючи обмін між ними. Якщо кешуються колективні дані, то вони реплікуються і можуть міститися в декількох кешах. Крім скорочення затримки доступу і необхідної смуги пропускання, така реплікація даних сприяє також загальному скороченню кількості обмінів.

Однак кешування розподілених даних викликає нову проблему – ***когерентність кеш-пам’яті***. Ця проблема виникає через те, що значення елемента даних в пам’яті, що використовується двома різними процесорами, є доступним цим процесорам тільки через їх індивідуальні кеші.

Для побудови більш потужних систем необхідні інші підходи. Одним з них є поділ пам’яті на незалежні модулі та забезпечення можливості доступу різних процесорів до різних модулів одночасно. Можливих рішень може бути багато, зокрема, використання матричного комутатора. Процесори і модулі пам’яті зв’язуються так, як показано на рис. 13.5. Три одночасно включених елемента, що дозволяють з'єднуватися наступним парам «центральний процесор – модуль пам'яті»: (C2, M1), (C3, M2) і (C4, M4).

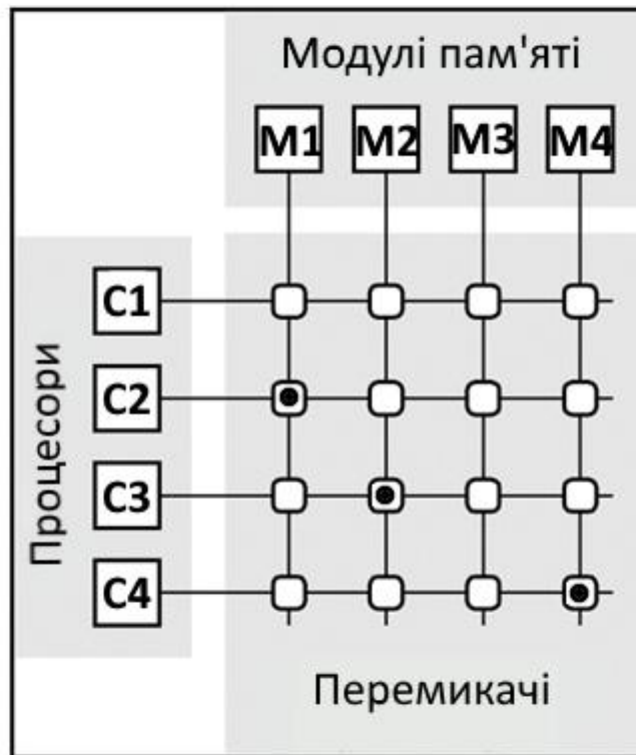


Рисунок 13.5 – Мультипроцесорна система з матричним комутатором

На перетині ліній розташовуються елементарні точкові перемикачі, що дозволяють або забороняють передачу інформації між процесорами і модулями пам’яті. Безумовною перевагою такої організації є можливість одночасної роботи процесорів з різними модулями пам’яті. Природно, шо в ситуації, коли два процесори захочуть працювати з одним модулем пам’яті, один з них буде заблокований.

Недоліком матричних комутаторів є великий обсяг необхідного обладнання, оскільки для зв’язку N процесорів з N модулями пам’яті потрібно N^2 елементарних перемикачів. У багатьох випадках це є занадто дорогим рішенням, що змушує розробників шукати інші шляхи.

13.5.2 Мультипроцесорні ОС

Тепер перейдемо від апаратного забезпечення мультипроцесорів до їх програмного забезпечення, зокрема до мультипроцесорних операційних систем. Можуть бути різні підходи до їх організації, три з яких будуть розглянуті далі. Слід зауважити, що всі ці підходи можна в рівній мірі застосувати як до багатоядерних систем, так і до систем, складених з окремих центральних процесорів.

Використання власної ОС для кожного центрального процесора. Найпростіший з можливих способів організації мультипроцесорної ОС полягає в статичному розподілі пам’яті на кілька розділів за кількістю наявних центральних процесорів і виділення кожному центральному процесору власної пам’яті і своєї копії ОС (рис. 13.6) [9].

Рисунок 13.6 – Поділ пам'яті між чотирма CPU із загальною копією коду ОС

Фактично *N* центральних процесорів працюють як *N* незалежних комп'ютерів. Очевидний варіант оптимізації – це дозволити всім CPU спільно використовувати код ОС і зберігати тільки індивідуальні копії даних, Квадратики, помічені словом Data, означають персональні дані ОС для кожного CPU.

Така схема краще, ніж *N* незалежних комп'ютерів, так як вона дозволяє всім машинам спільно використовувати набір дисків та інших пристроїв введення-виведення, а також забезпечує гнучке спільне використання пам'яті.

Наприклад, якщо потрібно запустити велику програму, одному з CPU може бути виділена велика порція пам'яті на час виконання цієї програми. Крім того, процеси можуть ефективно спілкуватися один з одним, якщо одному процесу буде дозволено писати дані в пам'ять, а інший процес буде їх зчитувати в цьому місці.

З точки зору операційних систем наявність ОС у кожного CPU є вкрай примітивним підходом. Слід зазначити три основні недоліка даної схеми мультипроцесорної ОС:

1. Коли процес звертається до системного виклику, то системний виклик перехоплюється і обробляється його власним CPU за допомогою структур даних в таблицях ОС.
2. Оскільки у кожної ОС є свої власні таблиці, у неї є також і свій набір процесів, які вона сама планує. Спільного використання процесів немає. Якщо користувач реєструється на CPU 1, то всі його процеси працюють на CPU 1. У результаті може статися так, що CPU 1 виявиться завантажений роботою, тоді як CPU 2 буде простоювати.
3. Спільного використання сторінок також немає. Може трапитися так, що у CPU 2 багато вільних сторінок, в той час як CPU 1 буде постійно займатися свопінгом. І немає ніякого способу зайняти вільні сторінки в сусіднього CPU, так як виділення пам'яті статично фіксоване.

З причини, наведених вище недоліків, така модель тепер використовується нечасто, хоча вона застосовувалася на зорі епохи мультипроцесорів, коли ставили за мету просто перенести існуючі ОС на будь-який новий мультипроцесор якомога швидше.

Мультипроцесори, що працюють за схемою «головний-підлеглий». Друга модель показана на рис. 13.7 [9]. Тут використовується одна копія операційної системи, а її таблиці існують виключно для центрального процесора 1. Всі системні виклики переадресовуються для подальшої обробки центральному процесору 1.

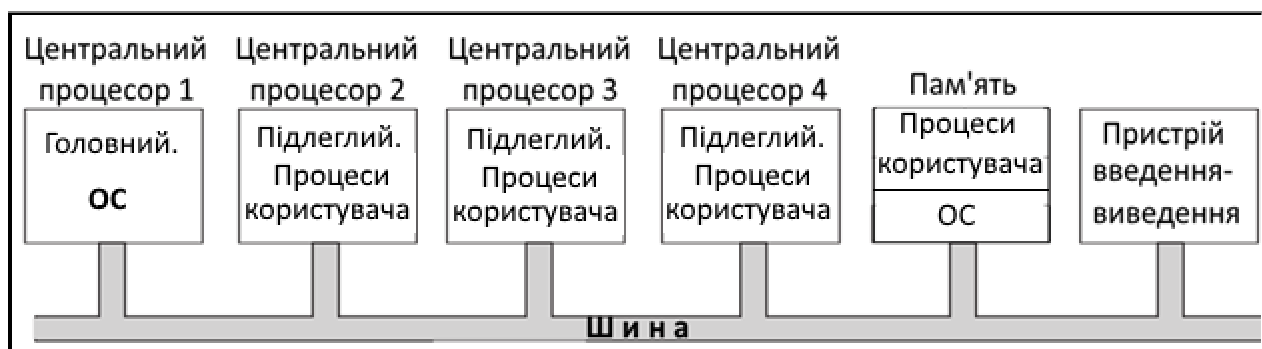


Рисунок 13.7 – Мультипроцесорна модель «головний-підлеглий»

Цей центральний процесор, якщо йому на це вистачає часу, може також запускати процеси користувача. Ця модель називається «головний-підлеглий», тому що центральний процесор 1 є головним, а всі інші – підлеглими.

Модель мультипроцесора «головний-підлеглий» дозволяє вирішити більшість проблем першої моделі.

У цій моделі використовується єдина структура даних (наприклад, один загальний список або набір пріоритетних списків), що враховує готові процеси. Коли CPU переходить в стан простою, він запитує в ОС процес, який можна обробляти, і при наявності готових процесів ОС призначає цьому CPU процес. Тому при такій організації: ніколи не може статися так, що один CPU буде простоювати, в той час як інший CPU перевантажений. Сторінки пам’яті можуть динамічно надаватися всім процесам.

Недолік такої моделі: при великій кількості центральних процесорів CPU-господар може стати вузьким місцем системи. Адже йому доводиться обробляти системні переривання від усіх CPU. Наприклад, якщо обробка системного переривання займає 10% часу, тоді 10 центральних процесорів дадуть йому граничне навантаження, а при 20 процесорах головний процесор вже не буде встигати їх обробляти, і система почне простоювати. Отже, така модель проста і працездатна для невеликої кількісті мультипроцесорів, але з великою кількістю процесорів вона працювати ефективно не може.

Симетричні мультипроцесори. Третя модель, що представляє собою симетричні мультипроцесори (SMP), дозволяє усунути перекіс попередньої моделі. Як і в попередній схемі, в пам’яті знаходиться всього одна копія ОС, але виконувати її може будь-який CPU. При системному виклику на CPU, яка звернулася до системи з системним викликом, відбувається переривання з переходом в режим ядра і обробкою системного виклику (рис. 13.8) [9].

Рисунок 13.8 – Модель симетричного мультипроцесора

Ця модель забезпечує динамічний баланс процесів і пам’яті, оскільки в ній є всього один набір таблиць ОС. Вона також дозволяє уникнути простою системи, пов’язаного з перевантаженням ведучого (головного) CPU, так як в ній немає ведучого CPU.

І все ж ця модель має власні проблеми. Зокрема, якщо два CPU одночасно беруть один і той же процес для запуску або запитують одну і ту ж вільну сторінку пам’яті, може статися катастрофа. Найпростіший спосіб вирішення подібних проблем полягає в зв’язуванні мьютекса (тобто блокування) з ОС, в результаті чого вся система перетворюється в одну велику критичну область. Коли CPU хоче виконувати код ОС, він повинен спочатку отримати мьютекс. Якщо мьютекс заблокований, CPU змушений чекати. Таким чином, будь-який CPU може виконати код ОС, але в кожен момент часу тільки один з них буде робити це.

13.5.3 Планування роботи мультипроцесора

На мультипроцесорі планування двовимірне. Планувальник повинен вирішити, який процес і на якому CPU запустити. Цей додатковий вимір істотно ускладнює планування на мультипроцесорах.

Інший ускладнюючий фактор полягає в тому, що в деяких системах всі процеси є незалежними, тоді як в інших системах вони формують групи залежних процесів.

Прикладом першої ситуації є система розподілу часу, в якій незалежні користувачі запускають незалежні процеси. Ці процеси не пов’язані один з одним, і планування кожного з них не залежить від інших процесів.

Приклад другий ситуації часто зустрічається в середовищі розробки великих програм. Великі системи, як правило, складаються з деякої кількості заголовків файлів, що містять макроси, визначення типів і оголошення змінних, які використовуються у файлах програми.

Розподіл часу. Розглянемо спочатку випадок планування незалежних процесів. Найпростіший алгоритм планування незалежних процесів (або потоків) полягає в підтримці єдиної структури даних для готових процесів, можливо, просто списку, але швидше за все множини списків для процесів з

різними пріоритетами (рис. 13.9, *а*) [9]. Тут всі 16 CPU в даний момент зайняті, а 14 процесів з різними пріоритетами очікують запуску.

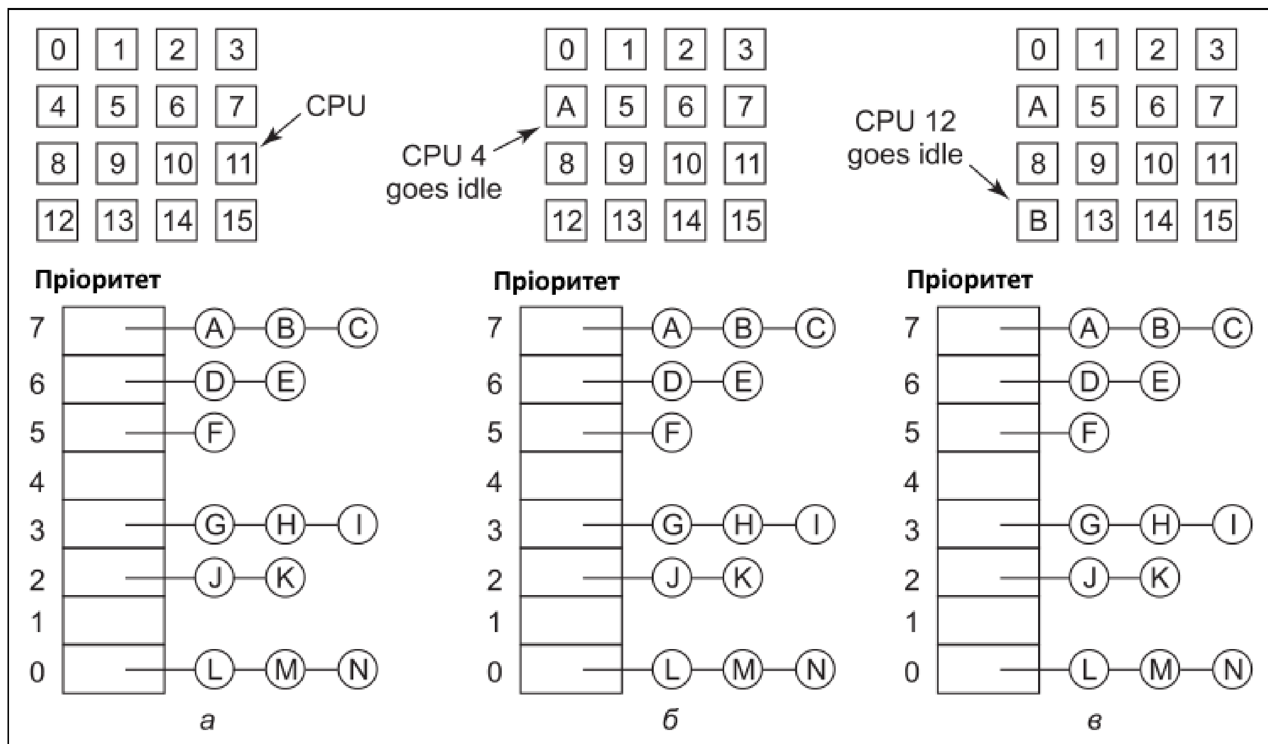


Рисунок 13.9 – Використання єдиної структури даних для планування мультипроцесора

Першим закінчує роботу (або його процес блокується) CPU 4. При цьому він блокує чергу планування і вибирає з неї процес з найвищим пріоритетом, тобто процес A (рис. 13.9, *б*). Потім звільняється CPU 12 і вибирає процес B (рис. 13.9, *в*). Поки ці процеси незалежні, подібне планування являє собою розумний вибір.

Наявність єдиної структури даних планування, яка використовується всіма процесорами, забезпечує процесору режим розподілу часу подібно до того, як це виконується на однопроцесорній системі. Крім того, така організація дозволяє автоматично балансувати навантаження, тобто вона виключає ситуацію, при якій один процесор простоює, в той час як інші процесори перевантажені.

Відзначимо два недоліки такої схеми планування:

– потенційне зростання конкуренції за структуру даних планування в міру збільшення числа CPU;
– звичайні накладні витрати на виконання перемикання контексту, коли процес блокується, чекаючи виконання операції введення-виведення.

Перемикання контексту також може статися, коли закінчується квант часу процесу. Припустимо, що процес утримує спін-блокування. Спін-блокування є взаємним блокуванням пристрою, яке може мати тільки два значення: «заблоковано» і «розблоковано». Тому інші CPU, які чекають звільнення блокування, просто втрачають час в циклах очікування, поки цей процес не буде запущений знову і не відпустить блокування.

На однопроцесорних системах спін-блокування практично не застосовується. Тому, якщо процес, що утримує мьютекс, блокується і запускається інший процес, то при спробі отримати мьютекс другий процес буде тут же заблокований, і багато часу втрачено не буде.

Щоб вирішити дану проблему, в деяких системах застосовується ***розумне планування*** (smart scheduling), в якому процес, захоплюючий спин-блокування, встановлює прапор, який демонструє, що він в даний момент володіє спін-блокуванням. Коли процес звільняє блокування, він також очищає і прапор. Таким чином, планувальник не зупиняє процес, що утримує спін-блокування, а, навпаки, дає йому ще трохи часу, щоб той завершив виконання критичної області та відпустив мьютекс.

Спільне використання простору. Інший підхід до планування мультипроцесорів може бути використаний, якщо процеси пов'язані один з одним будь-яким способом. Будемо називати плановані об'єкти потоками, але все сказане тут справедливо і для процесів. Планування декількох потоків на декількох CPU називається ***спільним використанням простору*** або ***розподілом простору***.

Найпростіший алгоритм розподілу простору працює наступним чином. Припустимо, що відразу створюється ціла група пов'язаних потоків. У момент їх створення планувальник перевіряє, чи є вільні CPU за кількістю створюваних потоків. Якщо вільних CPU досить, кожному потоку виділяється власний CPU, тобто CPU працює в однозадачному режимі, і всі потоки запускаються. Якщо CPU недостатньо, то жоден з потоків не запускається, поки не звільниться достатня кількість CPU. Кожен потік виконується на своєму CPU аж до завершення, після чого все CPU повертаються в пул вільних CPU.

Якщо потік виявляється заблокованим операцією введення-виведення, він продовжує утримувати CPU, який простоює до тих пір, поки потік не зможе продовжувати свою роботу. При появі наступного пакета потоків застосовується той же алгоритм.

У будь-який момент часу декілька CPU статично розділяється на кілька підмножин, на кожному з яких виконуються потоки одного процесу. На рис. 13.10 показані підмножини з 6, 8 і 16 CPU, і 2 CPU залишилися не включеними в підмножини [9]. Згодом, у міру завершення роботи одних процесів і появи нових процесів, кількість і розміри груп CPU змінюються.

У цій простій моделі розбиття CPU на групи процес просто запитує певну кількість CPU і або відразу отримує їх, або чекає, поки вони не звільняться.

Один із способів активного управління ступенем розпаралелювання процесів полягає в наявності центрального сервера, який веде облік працюючих і охочих працювати процесів, а також мінімального і максимального кількості потрібних для них CPU. Періодично кожен CPU опитує центральний сервер, щоб дізнатися, скільки CPU він може використовувати. Потім він збільшує або зменшує кількість процесів або потоків, намагаючись домогтися відповідності числа доступних CPU.

Рисунок 13.10 – Набір з 32 CPU, розділений на чотири групи

Бригадне планування. Явною перевагою спільного використання простору є виняток багатозадачності, що знижує накладні витрати по переключенню контексту. Однак її недолік полягає у втрати часу при блокуванні CPU.

Щоб зрозуміти, які можливі проблеми при незалежному плануванні потоків процесу (або процесів завдання), розглянемо систему з потоками A_0 і A_1, що належать процесу A, і потоками B_0 і B_1, що належать процесу B. Потоки A_0 і B_0 працюють в режимі розподілу часу на CPU 0, а потоки A_1 і B_1 – на CPU 1. Потокам A_0 і A_1 потрібно часто обмінюватися інформацією.

Спілкування потоків виглядає наступним чином. Потік A_0 посилає потоку A_1 повідомлення, після чого потік A_1 відправляє потоку A_0 відповідь і т. д. Припустимо, що потоки A_0 і B_1 почали виконуватися першими, як показано на рис. 13.11 [9].

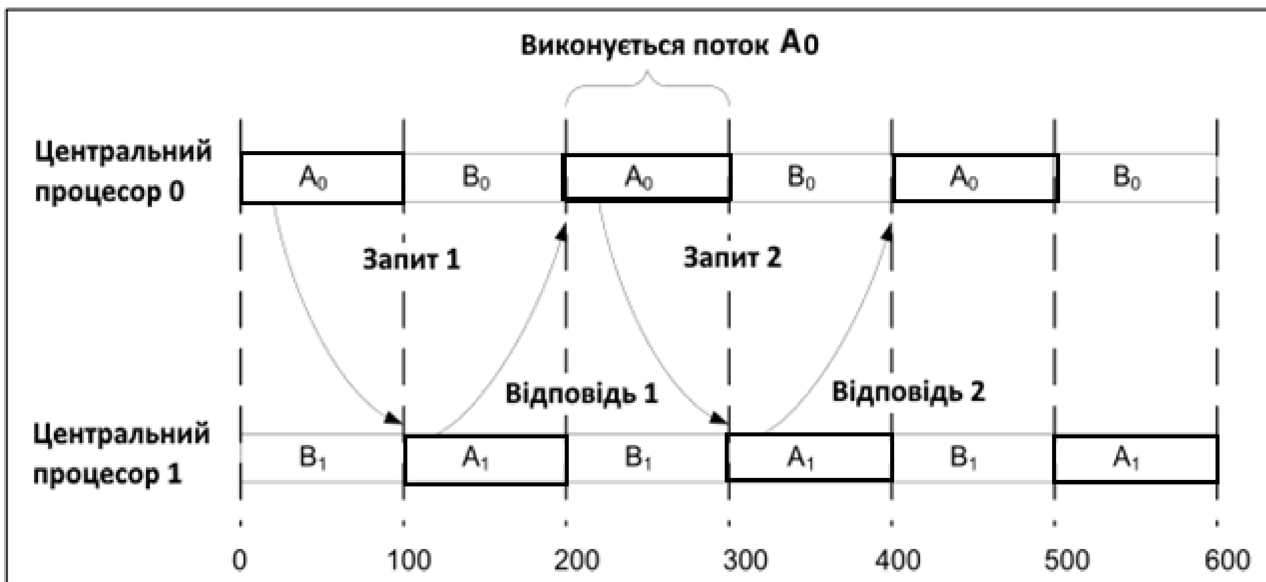


Рисунок 13.11 Спілкування двох потоків, що належать процесу *A*

Рішенням даної проблеми є так зване ***бригадне планування***, що представляє собою розвиток ідеї спільного планування. Бригадне планування складається з трьох частин:

1. Групи пов’язаних потоків плануються як одне ціле – бригада.
2. Всі члени бригади запускаються одночасно на різних центральних процесорах, що працюють в режимі розподілу часу.
3. У всіх членів бригади кванти часу починаються і закінчуються одночасно.

Бригадне планування працює завдяки синхронності роботи всіх CPU. Це означає, що час розділяється на дискретні кванти, як було показано на рис. 13.11. На початку кожного нового кванта всі CPU переплановуються заново, і на кожному CPU запускається новий потік. На початку наступного кванта знову приймається рішення про планування. В середині кванта планування не виконується. Якщо який-небудь потік блокується, його CPU простоює до кінця кванта часу. Приклад роботи бригадного планування наведено на рис. 13.12.

Часовий інтервал	Центральний процесор 0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Рисунок 13.12 – Бригадне планування

Тут показаний мультипроцесор з шістьма CPU, на яких працюють п’ять процесів, від *A* до *E*, із загальним числом потоків, рівним 24. Протягом часового інтервалу 0 потоки від A_0 до A_5 плануються і виконуються. Під час інтервалу 1 плануються і виконуються B_0-B_2 і C_0-C_2. Протягом часового інтервалу 2 плануються і виконуються п’ять потоків процесу *D* і потік E_0. Решта шість потоків процесу *E* працюють під час інтервалу 3. Потім цикл повторюється так, що часовий інтервал 4 повторює інтервал 0 і т. д.

Ідея бригадного планування полягає в тому, щоб всі потоки процесу працювали по можливості разом, так, якщо один із потоків посилає повідомлення іншому потоку, то другий потік отримує повідомлення практично миттєво і може так само швидко на нього відповісти. Оскільки всі потоки процесу *A* працюють разом протягом одного кванта часу, вони можуть відправляти і приймати велику кількість повідомлень за один квант часу, усуваючи, таким чином, проблему, зображену на рис. 13.11.

13.6 Мультипроцесорні системи з передачею повідомлень

Обчислювальні системи, як потужні засоби обробки завдань користувачів, широко використовуються не тільки автономно, але і в мережах ЕОМ в якості серверів.

Зі збільшенням розмірів мереж і їх розвитком зростають щільності інформаційних потоків, навантаження на засоби доступу до мережевих ресурсів і на засоби обробки завдань. Коло задач, що розв’язуються серверами, постійно розширюється, стає різноманітним і складним. Чим вище ранг мережі, тим більше спеціалізованими вони стають. Адміністратори мереж повинні постійно нарощувати їх міць і кількість, оптимізуючи характеристики мережі під зростаючі запити користувачів.

У мережах перших поколінь сервери будувалися на основі великих і дуже дорогих ЕОМ (mainframe), що випускаються цілою низкою компаній. Всі вони працювали під управлінням ОС Unix і здатні були об’єднуватися для спільної роботи.

Як і у всякій технології, що розвивається, складні універсальні сервери різних фірм-виробників повинні були поступитися місцем стандартним масовим рішенням. Успіхи мікроелектроніки, широке поширення Internet/Intranet-технологій дозволили перейти до більш простих і дешевих систем. Досвід створення серверів на основі SMP- і MPP-структур показав, що вони не забезпечують хорошої адаптації до конкретних умов функціонування, залишаються дорогими і складними в експлуатації.

Популярність і привабливість мультипроцесорів пов’язана з тим, що вони пропонують просту модель обміну даними, при якій всі центральні процесори спільно використовують загальну пам’ять. Єдина складність полягає в складності побудови великих мультипроцесорних систем і, як наслідок, їх дорожнечі. А дуже великі системи неможливо створити ні за яку ціну. Тому при необхідності масштабування на велику кількість центральних процесорів потрібно щось ще інше.

Одним з перспективних напрямків тут є ***кластеризація***, тобто технологія, за допомогою якої кілька серверів, самі є обчислювальними системами, об’єднуються в єдину систему більш високого рангу для підвищення ефективності функціонування системи в цілому. Цілями побудови кластерів служать:

- поліпшення масштабованості (здатність до нарощування потужності);
- підвищення надійності та готовності системи в цілому;
- збільшення сумарної продуктивності;
- ефективний перерозподіл навантажень між комп’ютерами кластера;
- ефективне управління і контроль роботи системи і т.п.

Поліпшення масштабованості, Або здатність до нарощування потужності передбачає, що всі елементи кластера мають апаратну, програмну та інформаційну сумісність. У поєднанні з простим і ефективним управлінням зміна обладнання в ідеальному кластері має забезпечувати відповідну зміну значень основних характеристик: додавання нових процесорів і дискових систем

повинно супроводжуватися пропорційним зростанням продуктивності, надійності тощо. У реальних системах ця залежність має нелінійний характер.

Масштабованість SMP- і MPP-структур досить обмежена. При великому числі процесорів в SMP-структурах зростає число конфліктів при зверненні до спільної пам'яті, а в MPP-структурах погано розв'язуються задачі перетворення і розбиття додатків на окремі завдання процесорам. У кластерах адміністратори мереж отримують можливість збільшувати пропускну здатність мережі за рахунок включення в нього додаткових серверів, навіть вже з числа працюючих, з урахуванням того, що балансування і оптимізація навантаження будуть виконуватися автоматично.

Наступною важливою метою створення кластеру є підвищення надійності та готовності системи в цілому. Саме ці якості сприяють популярності і розвитку кластерних структур. Надмірність, спочатку закладена в кластери, здатна їх забезпечити. Основою цього є можливість кожного сервера кластера працювати автономно, але в будь-який момент він може переключитися на виконання робіт іншого сервера в разі його відмови.

Збільшення сумарної продуктивності кластера, що об'єднує кілька серверів, зазвичай не є самоціллю, а забезпечується автоматично. Адже кожен сервер кластера сам є досить потужною обчислювальною системою, розрахованою на виконання ним усіх необхідних функцій в частині управління відповідними мережевими ресурсами. З розвитком мереж все більшого значення набувають і розподілені обчислення. При цьому багато комп'ютерів, в тому числі і сервери, можуть мати не дуже велике навантаження. Вільні ресурси домашніх комп'ютерів, робочих станцій локальних обчислювальних мереж та й самих серверів можна використовувати для виконання будь-яких трудомістких обчислень. При цьому вартість створення подібних обчислювальних кластерів дуже мала, так як всі їх складові частини працюють в мережі і тільки при необхідності утворюють віртуальний (тимчасовий) обчислювальний комплекс.

Сукупні обчислювальні потужності кластерів можуть бути порівнянні з потужностями супер-ЕОМ і навіть перевищувати їх при незмірно меншій вартості. Такі технології стосовно до окремих класів задач добре відпрацьовані. Коло таких задач не дуже широке, але число одночасно залучених для цих цілей комп'ютерів може бути величезним: десятки, сотні і навіть тисячі.

Робота кластера під керуванням єдиної операційної системи дозволяє оперативно контролювати процес обчислень і ефективно перерозподіляти навантаження на комп'ютери кластера.

Управління такими проектами вимагає створення спеціального клієнтського і серверного програмного забезпечення, що працює у фоновому режимі. Комп'ютери при цьому періодично отримують завдання від сервера, включаються в роботу і повертають результати обробки. Останні версії браузерів ще більш спрощують процес взаємодії, так як на клієнтській машині можна активізувати виконання різних програм-сценаріїв (скриптів).

Ефективне управління і контроль роботи системи має на увазі можливість роботи окремо з кожним вузлом, відключення вручну або програмно його для модернізації або ремонту з подальшим поверненням його в працюючий

кластер. Ці операції приховані від користувачів, які просто не помічають їх. Кластерне ПЗ, інтегроване в операційні системи серверів, дозволяє працювати з вузлами як з єдиним пулом ресурсів (Single System Image – SSI), вносячи необхідні загальні зміни за допомогою однієї операції для всіх вузлів.

Кластери об'єднують кілька серверів під єдиним управлінням. Всі нові сервери, як правило, є багатопроцесорними і відносяться до SMP-структурам, що забезпечує можливість багатоступінчастого перемикання навантаження елемента, що відмовив як всередині кластера, так і всередині сервера. Існують сервери з різною кількістю процесорів (від двох до шістнадцяти). Фірма Sun працює навіть над створенням 64-процесорної SMP-моделі сервера. IBM передбачає з появою мікропроцесора Itanium II випустити SMP-систему, розраховану на 16 процесорів. Навпаки, фірма Dell вважає, що застосування більш восьми процесорів в SMP-структурі застосовувати недоцільно через труднощі подолання конфліктів при зверненні їх до загальної оперативної пам'яті [9].

Великий інтерес до побудови кластерів проявляє фірма Microsoft. У зв'язку широкою популярністю операційних систем сімейства Windows NT, призначених для управління мережами великих підприємств, з'явилися різні варіанти кластерного забезпечення. Передбачається, що воно буде підтримувати до 16 і більше вузлів в кластері.

Уніфікація інженерно-технічних рішень передбачає відповідно і стандартизацію апаратних і програмних процедур обміну даними між серверами. Для передачі керуючої інформації в кластері використовуються спеціальні магістралі, які мають більш високі швидкості обміну даними. В якості такого стандарту пропонується інтелектуальне введення-виведення (Intelligent Input / Output, I2O). Специфікація I2O визначає уніфікований інтерфейс, звільняючи процесори і їх системні шини від обслуговування периферії [9].

Як і в будь-якої нової технології, у кластеризації є свої недоліки:

- затримки розробки і прийняття загальних стандартів;
- велика частка нестандартних і закритих розробок різних фірм, що утруднюють їх спільне використання;
- проблеми управління одночасним доступом до файлів;
- складності з керуванням конфігурацією, налаштуванням, розгортанням, оповіщенням серверів про збої тощо.

Планування мультикомп'ютерів. У мультипроцесорах всі процеси перебувають в одній і тій же пам'яті. Коли центральний процесор завершує своє поточне завдання, він вибирає процес і запускає його. В принципі, потенційними кандидатами є всі процеси. На мультикомп'ютері ситуація зовсім інша. У кожного вузла є власна пам'ять і власний набір процесів. Центральний процесор 1 не може раптово прийняти рішення про запуск процесу на вузлі 4, не виконавши спочатку деякої кількості роботи, щоб отримати цей процес. Це означає, що планування на мультикомп'ютерах здійснюється простіше, а розподіл процесів по вузлах набуває більш важливу роль.

Планування мультикомп'ютерних систем чимось нагадує планування мультипроцесорних систем, але не всі колишні алгоритми можуть бути

застосовані до цього планування. Але найпростіший алгоритм, застосовуваний на мультипроцесорах, – ведення єдиного централізованого списку готових процесів – в даному випадку не працює, тому що кожен процес може виконуватися лише на тому центральному процесорі, в пам’яті якого в даний момент він знаходиться. Проте при створенні нового процесу можна вибирати, куди його помістити, для того щоб, наприклад, збалансувати навантаження.

Оскільки в кожного вузла є власні процеси, може бути застосований будь-який локальний алгоритм планування. Разом з тим можна застосувати бригадне планування, як і на мультипроцесорах, оскільки для нього потрібна лише початкова угода про те, який процес в якому кванті часу буде працювати [9].

Контрольні питання і тести до розділу 13

Контрольні питання

1. Дайте визначення сильнозв'язаним багатопроцесорним системам.
2. Які три основні задачі включає планування в багатопроцесорній системі?
3. Дайте визначення статичного і динамічного призначення процесорів процесам.
4. Які ключові функції ОС виконуються при призначенні процесів процесорам при підході «головний-підлеглий»?
5. Які ключові функції ОС виконуються при призначенні процесів процесорам при підході «рівноправні процесори»?

Тести

1. Спосіб організації обчислювального процесу в системах з декількома процесорами називається:
   1) мультизадачна обробка;
   2) мультипроцесорна обробка;
   3) мультипрограмна обробка;
   4) мультипроцесна обробка.
2. Якщо система складаються з набору відносно автономних комп'ютерів, кожен процесор якої має власну основну пам'ять і канали введення-виведення, то таку багатопроцесорну систему можна класифікувати як:
   1) слабкозв'язану систему;
   2) сильнозв'язану систему;
   3) розподілену систему.
3. Якщо система складається з набору процесорів, які спільно використовують загальну основну пам'ять і знаходяться під загальним управлінням ОС, то таку багатопроцесорну систему можна класифікувати як:
   1) слабкозв'язану систему;
   2) сильнозв'язану систему;
   3) розподілену систему.

4. Якщо система складається з набору незалежних комп'ютерів, який представляється їх користувачам єдиною об'єднаною системою, то таку багатопроцесорну систему можна класифікувати як:
   1) слабкозв'язану систему;
   2) сильнозв'язану систему;
   3) розподілену систему.
5. Виберіть принципову відмінність між клієнтом і сервером в мережі:
   1) ініціатором виконання роботи мережевою службою завжди виступає клієнт, а сервер завжди знаходиться в режимі активного очікування запитів;
   2) ініціатором виконання роботи мережевою службою завжди виступає клієнт, а сервер завжди знаходиться в режимі пасивного очікування запитів;
   3) ініціатором виконання роботи мережевою службою завжди виступає сервер, а клієнт завжди знаходиться в режимі пасивного очікування для формування запитів.
6. У разі відмови одного з процесорів **...** системи, як правило, легко реконфігуруються, що є їх великою перевагою.
   1) асиметричні;
   2) симетричні.
7. При бригадному плануванні:
   1) групи пов’язаних потоків плануються як одне ціле (бригада) для одного центрального процесора;
   2) всі члени бригади запускаються одночасно на різних центральних процесорах;
   3) всі члени бригади запускаються по черзі на різних центральних процесорах.

14 ПЛАНУВАННЯ РЕАЛЬНОГО ЧАСУ

14.1 Поняття систем реального часу

Обчислення в реальному часі стають усе реальнішою галуззю знань. Операційна система, і зокрема планувальник, є найважливішим компонентом системи реального часу. Системи реального часу застосовуються для управління різними технічними об'єктами, такими, наприклад, як верстат, супутник, телекомунікації, інтелектуальні системи управління, управління космічними і підводними станціями. В усіх цих випадках існує гранично допустимий час, впродовж якого має бути виконана та або інша програма, що управляє об'єктом, інакше може статися аварія (наприклад, супутник вийде із зони видимості).

Чим принципово відрізняються операційні системи реального часу від ОС загального призначення? Операційні системи загального призначення, особливо багатокористувацькі, орієнтовані на оптимальний розподіл ресурсів комп'ютера між користувачами і задачми (системи розподілу часу), В операційних системах реального часу подібна задача відходить на другий план – все відступає перед головною задачою – встигнути зреагувати на події, що відбуваються на об'єкті.

Інша відмінність – застосування операційної системи реального часу завжди пов'язане з апаратурою, з об'єктом, з подіями, що відбуваються на об'єкті. Система реального часу, як апаратно-програмний комплекс, що включає в себе датчики, які реєструють події на об'єкті, модулі введення-виведення, що перетворюють показния датчиків в цифровий вигляд, придатний для обробки цих показань на комп'ютері, і, нарешті, комп'ютер з програмою, що реагує на події, що відбуваються на об'єкті. Операційна система реального часу орієнтована на обробку зовнішніх подій. Саме це призводить до корінних відмінностей в структурі ОС загального призначення.

Канонічне визначення системи реального часу дано Дональдом Гіллієсом [12]: «Системою реального часу є така система, коректність функціонування якої визначається не тільки коректністю виконання обчислень, але і часом, за який отримано необхідний результат. Якщо вимоги за часом не виконуються, то вважається, що відбулася відмова системи».

Таким чином, критерієм ефективності для систем реального часу є їх здатність витримувати заздалегідь задані інтервали часу між запуском програми і отриманням результату (керуючої дії). Обчислення в реальному часі – тип обчислень, в яких коректність системи залежить не лише від логічного результату обчислень, але і часу отримання цього результату. Цей час називається часом реакції системи, а відповідна властивість системи – ***реактивністю***. Для цих систем мультипрограмна суміш є фіксованим набором заздалегідь розроблених програм, а вибір програми на виконання здійснюється виходячи з поточного стану об'єкта або відповідно до розкладу планових робіт.

По суті, частина цих задач є задачами реального часу, у кожній з яких своя міра терміновості. Такі задачі намагаються управляти подіями або реагувати на події, що відбуваються в зовнішньому світі. Оскільки ці події відбуваються в реальному часі, то і задачі, пов'язані з ними, не повинні відставати від реальних

подій. Такі задачі можна охарактеризувати як жорсткі і м'які. ***Жорсткі задачі реального часу*** (hard real-time task) є завданнями, які повинні відповідати цим граничним термінам; інакше неминучі фатальні збої системи. ***М'які задачі реального часу*** (soft real-time task) також мають граничні терміни, але їх виконання – швидше побажання, ніж обов'язок. Якщо навіть така задача перевищило відведений їй час, її все одно варто продовжувати планувати і довести до завершення.

Таким чином, для того щоб система могла задовольняти вимогам, що пред'являються до систем реального часу, апаратні, програмні засоби та алгоритми роботи системи повинні гарантувати задані тимчасові параметри реакції системи. Час реакції не обов'язково має бути дуже маленьким, але він повинен бути гарантованим і таким, що відповідає поставленим вимогам.

До недавнього часу для вирішення задач автоматизації в основному використовувалися операційні системи реального часу, засновані на микроядерной архітектурі, – такі як VxWorks, OS-9, PSOS, QNX, Windows NT Embedded, LynxOS. Ці системи володіють швидким часом реакції на події і переривання, компактністю коду, хорошою вбудовуваністью та іншими перевагами, характерними для операційних систем з микроядерною архітектурою.

В даний час відбувається активний процес злиття універсальних ОС і ОС реального часу. На програмному ринку з'являються різні інструменти підтримки режиму реального часу, вбудовані в звичні операційні системи. Цей клас продуктів володіє значними перевагами з боку користувачів і програмістів, поєднуючи в собі звичний інтерфейс, засоби розробки програм і API-інтерфейс реального часу.

14.2 Характеристики ОС реального часу

Операційні системи реального часу повинні задовольняти таким вимогам:

- детермінізм;
- чутливість;
- управління з боку користувача;
- надійність;
- відновлення після збоїв.

ОС ***детермінована***, якщо вона виконує операції у фіксований, зумовлений час або в межах зумовлених інтервалів часу. При конкуренції процесів за володіння ресурсами і часом процесора система не може бути повністю детермінованою. Наскільки детерміновано система здатна задовольняти запити, залежить в першу чергу від швидкості, з якою вона здатна реагувати на переривання, а також від того, чи має система достатню пропускну спроможність для обробки всіх запитів за необхідний час.

Схожою з детермінізмом характеристикою є ***чутливість*** системи. Детермінізм зосереджений на часі затримки перед розпізнаванням переривання. Чутливість розглядає питання про те, скільки часу потрібно ОС для обслуговування переривання після його розпізнавання. Детермінізм і чутливість

у сукупності утворюють час відгуку на зовнішню подію. Вимоги до часу відгуку є критичними для систем реального часу.

Управління з боку користувача в системах реального часу значно ширше, ніж у звичайній ОС. У звичайній ОС користувач або не в змозі управляти функцією планування ОС, або може здійснювати найзагальніше керівництво типу угрупування користувачів за декількома класами пріоритетів. Проте в системах реального часу важливою складовою є забезпечення можливості тонкого налаштування пріоритетів задач. Користувач повинен мати можливість розділяти задачі на жорсткі і м'які і визначати відносні пріоритети в межах кожного класу. Крім того, системи реального часу дозволяють користувачеві визначати і такі характеристики, як використання сторінкової організації пам'яті або свопінг процесів, а також визначати, які процеси повинні знаходитися постійно в основній пам'яті, які права процесів з різних груп і багато чого іншого.

Надійність в системах реального часу – дуже важливе питання. Випадкова помилка в звичайній ОС може бути в гіршому разі оброблена просто перезавантаженням системи. Збій одного з процесорів у багатопроцесорній системі призведе до зниження рівня обслуговування, до заміни або налагодження цього процесора. Але система реального часу працює, як і виходить з її назви, з подіями в реальному часі, і втрата продуктивності може призвести до катастрофічних наслідків

Відновлення після збоїв – це характеристика системи, яка описує здатність системи зберегти максимальну функціональність і не втратити дані при збої. Системи реального часу намагаються або виправити ситуацію повністю, або мінімізувати її вплив на тривалу роботу системи.

Для того щоб відповідати певним вимогам, сучасна система реального часу повинна включати:

- швидке перемикання процесів і/або потоків;
- малий розмір (мінімальна функціональність);
- багатозадачність із засобами взаємодії процесів (такими, як семафори, сигнали, події);
- витісняюче планування на основі системи пріоритетів;
- мінімізація періодів часу, коли переривання заборонені.

14.3 Планування реального часу

Важливою частиною будь-якої ОС реального часу є планувальник завдань. Незважаючи на те, що в різних джерелах він може називатися по-різному (диспетчер задач, супервізор тощо), його функції залишаються тими ж: визначити, яка з задач повинна виконуватися в системі в кожен конкретний момент часу.

В системі з одним процесором ядро операційної системи повинно планувати доступ паралельних завдань до процесора. Ядро підтримує список готових до роботи завдань. Для призначення центрального процесора той чи іншій задачі було розроблено багато різних алгоритмів, в тому числі циклічне обслуговування і витісняюче планування з пріоритетами.

Для систем реального часу циклічне планування не годиться. Справедливий розподіл ресурсів – це не головне, задачам потрібно призначати пріоритети у відповідності з важливістю виконуваних операцій. Так, критичні за часом задачі обов'язково повинні вкластися у відведені часові рамки. Тому для систем реального часу більше підходить алгоритм яка витісняє планування з пріоритетами.

Спочатку алгоритми планування в реальному часі розроблялися для незалежних періодичних завдань, тобто таких періодичних завдань, які не взаємодіють один з одним і, отже, не потребують синхронізації. З тих пір було проведено багато теоретичних досліджень, результати яких можна застосовувати до багатьох практичних задач.

Періодична задача характеризується періодом F (частота запуску) і часом виконання T (час центрального процесора, необхідний для завершення одного запуску). Коефіцієнт використання центрального процесора для неї дорівнює $U = T/F$. Задача називається планованою, якщо вона задовольняє всім тимчасовим обмеженням, тобто її виконання завершується до закінчення періоду. Група задач іменується планованою, коли планованою є кожна задача, яка входить в цю групу.

Наведемо короткий огляд різних підходів до проблеми планування реального часу. Алгоритми планування для систем реального часу можуть бути як статичними, так і динамічними. При статичному плануванні усі рішення планування приймаються заздалегідь, ще до запуску системи. У разі динамічного планування рішення планування застосовуються по ходу справи.

Статичне планування з використанням таблиць. При цьому плануванні виконується статичний аналіз здійсненності планування, результатом якого є план, який в процесі роботи системи визначає, коли повинне початися виконання завдань.

Таке планування застосовне для періодичних завдань. Вхідною інформацією для аналізу є час появи завдань в системі, час виконання, граничні терміни виконання і відносний пріоритет кожного завдання. Планувальник намагається розробити такий план роботи, який би задовольняв усім часовим вимогам завдань. Такий підхід є передбачуваним, але абсолютно не гнучким, оскільки будь-яка зміна вимог будь-якого завдання призводить до необхідності перегляду усього розкладу.

Статичне витісняюче планування на основі пріоритетів. У цьому випадку також виконується статичний аналіз, але розклад не створюється. Замість цього на основі проведеного аналізу завданням призначаються пріоритети з тим, щоб далі можна було використати традиційний витісняючий планувальник, працюючий з урахуванням пріоритетів завдань.

Динамічне планування на основі розкладу. Планування визначається в процесі роботи (динамічно). Завдання, що поступило в систему, приймається тільки в тому випадку, якщо визначена можливість його виконання з урахуванням усіх часових вимог.

Після надходження завдання в систему (але до початку його виконання) робиться спроба створити розклад, який містить як усі наявні в системі завдання,

так і ті, що знову поступили. Якщо вдається створити такий розклад, що при виконанні задовольняються часові обмеження як нового завдання, так і вже наявних в системі завдань, воно приймається планувальником.

Динамічне планування найкращого результату. При цьому аналіз здійсненності планування не виконується. Система намагається задовольнити всі граничні терміни і знімає ті процеси, граничні терміни яких порушені.

Динамічне планування найкращого результату використовується в багатьох сучасних комерційних системах реального часу. Як правило, завдання неперіодичні, а тому непридатний статичний аналіз планування. При такому типі планування невідомо, чи будуть задоволені часові обмеження завдання до тих пір, поки завдання не буде повністю виконано (чи доки не будуть порушені часові обмеження). Саме це і є основним недоліком цієї схеми. Перевага же динамічного планування найкращого результату – в простоті реалізації.

Усі розглянуті підходи планування засновані на додатковій інформації.

1. ***Час готовності***. Час, коли завдання стає доступним для виконання. У завданнях, що повторюються або періодичних завданнях, час готовності є послідовністю заздалегідь відомого часу. У неперіодичному завданні цей час може бути відомий заздалегідь.
2. ***Граничний час початку виконання*** – це час, коли повинно початися виконання завдання.
3. ***Граничний час завершення виконання*** – це час, коли завдання має бути повністю завершене. Звичайне завдання реального часу має обмеження або за граничним часом початку виконання, або за граничним часом завершення виконання, але не обидва обмеження одночасно.
4. ***Час виконання –*** це час, якого потребує завдання для повного виконання. У деяких випадках цей час відомий, а в деяких система сама оцінює зважене середнє значення часу виконання.
5. ***Вимоги до ресурсів*** – це певна кількість ресурсів (відмінних від процесора), яких потребує завданням при його виконанні.
6. ***Пріоритет*** – це міра відносної важливості завдання. Жорсткі завдання мають «абсолютний» пріоритет, призводячи до збою системи при порушенні часових обмежень цих завдань.
7. ***Структура підзадач***. Завдання може бути розбите на обов'язкові і необов'язкові підзадачі. Жорсткі граничні терміни при такому розділенні мають тільки обов'язкові завдання.

Ще одним критичним питанням є ***витіснення***. Якщо визначається граничний час початку роботи, то має сенс застосування невитісняючого планування. В цьому випадку бажано, щоб завдання реального часу після завершення обов'язкової або критичної частини самостійно блокувалися, дозволяючи виконуватися іншим завданням.

Як приклад планування періодичних завдань з граничним часом завершення розглянемо систему, яка збирає і обробляє дані від датчиків A і B. Терміни збору даних від датчика A – кожні 20 мс, датчика B – кожні 50 мс.

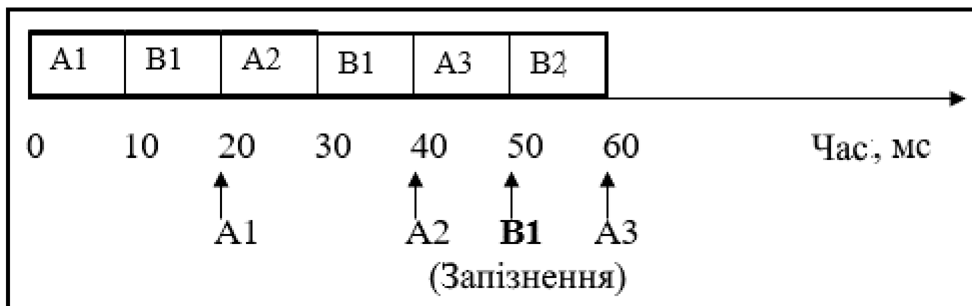
Процес зняття даних, включаючи накладні витрати ОС, займають для датчика А 10 мс, а для датчика В – 25 мс. У таблиці. 14.1 приведений профіль виконання цих двох завдань.

Таблиця 14.1 – Профіль виконання двох періодичних завдань

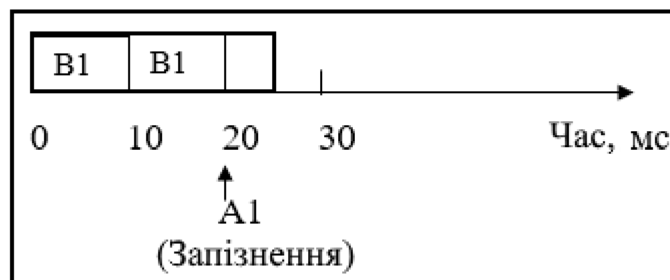
Процес	Час надходження	Час виконання	Граничний час закінчення
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
…	…	…	…
B(1)	0	25	50
B(2)	50	25	100
…	…	…	…

Комп'ютер може приймати рішення про планування кожні 10 мс.

Припустимо, що за цих умов ми використовуємо схему планування з пріоритетами. Якщо А має вищий пріоритет, завдання В1 отримає тільки 20 мс процесорного часу в двох суміжних інтервалах по 10 мс. Після цього буде досягнутий граничний час його виконання, і завдання виконане не буде.



Якщо вищий пріоритет отримає завдання В, то виконатися в строк не зможе завдання А1.



Якщо застосуємо в цій ситуації планування з найранішим граничним терміном, то у момент часу *t = 0* поступають завдання А1 і В1. Оскільки граничний термін А1 настає раніше граничного терміну В1, спершу виконується завдання А1. Після його завершення починається виконання завдання В1. У момент часу *t = 20* в систему поступає завдання А2, граничний термін виконання якого настає раніше граничного терміну виконання В1. Відповідно, завдання В1

переривається, і починається виконання завдання A2. Виконання завдання B1 триває, починаючи з моменту *t = 30*.

У момент *t = 40* в систему поступає завдання A3, граничний термін виконання якого пізніший, ніж граничний термін виконання завдання B1, так що завдання B1 продовжує виконуватися до завершення в момент часу *t = 45*. У цей момент починається виконання завдання A3, яке завершується до моменту *t = 55*.

A1, B1 | A2 | A3 | B2 | A4 | A5

A1 | B1 | A2 | B1 | B1 | A3 | A3 | B2 | A4 | B2 | B2 | A5

0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 t, мс

A1 | A2 | B1 | A3 | A4 | A5, B2

У наведеному прикладі, де в кожній точці витіснення планувальник дає вищий пріоритет тому завданню, граничний термін якого настає раніше, задовольняються всі вимоги до системи.

Контрольні питання і тести до розділу 14

Контрольні питання

1. Яке основне призначення ОС реального часу?
2. Який головний критерій ефективності для систем реального часу?
3. Що таке «реактивність» ОС реального часу?
4. Охарактеризуйте «м'які» і «жорсткі» задачі ОС реального часу.
5. Охарактеризуйте роботу ОС реального часу при статичному плануванні з використанням таблиць.
6. У чому різниця між динамічним плануванням на основі розкладу і статичним плануванням з використанням таблиць?
7. Назвіть основний недолік при динамічному плануванні найкращого результату.

Тести

1. Яке призначення системи реального часу?
    1) служить для управління яким-небудь об'єктом в реальному масштабі часу;
    2) служить для визначення точного часу в будь-який момент;
    3) служить для планування реального графіку використання робочого часу;
    4) служить для розв'язання реальних задач.
2. У яких системах гарантується виконання завдання за певний проміжок часу?
    1) у системах пакетної обробки;
    2) у системах розподілу часу;

3) у багатозадачних системах;
4) у системах реального часу.

3. Планувальник називається статичним, якщо він приймає рішення про планування:
   1) не під час роботи системи, а заздалегідь;
   2) під час роботи системи на основі статичного аналізу поточної ситуації;
   3) після появи завдання в системі (але до початку його виконання) робиться спроба створити розклад.
4. Спосіб організації обчислювального процесу в системах з декількома процесорами називається:
   1) мультизадачна обробка;
   2) мультипроцесорна обробка;
   3) мультипрограмна обробка;
   4) мультипроцесна обробка.
5. Здатність системи витримувати заздалегідь задані інтервали часу між запуском програми і отриманням результату називається:
   1) реактивністю;
   2) детермінізмом;
   3) чутливістю.
6. У системах реального часу жорсткі завдання мають:
   1) «абсолютний» пріоритет;
   2) «відносний» пріоритет;
   3) «абсолютний» і «відносний» пріоритет.
7. До якого різновиду систем відноситься система, що відстежує зміни температури на атомній електростанції?
   1) до «м'якої» системи реального часу;
   2) до «жорсткої» системи реального часу;
   3) до системи розподілу часу.

15 УПРАВЛІННЯ ВВЕДЕННЯМ-ВИВЕДЕННЯМ

Одним з головних задач ОС є забезпечення обміну даними між додатками і периферійними обладнанням комп'ютера. Заради виконання цієї задачі і були розроблені перші системні програми, які послужили прототипами операційних систем. У сучасній ОС функції обміну даними з периферійними пристроями виконує підсистема введення-виведення. Клієнтами цієї підсистеми є не лише користувачі і додатки, але і деякі компоненти самої ОС, яким потрібне отримання системних даних або їх виведення. Наприклад, підсистемі управління процесами при зміні активного процесу необхідно записати на диск контекст призупиненого процесу і зчитати з диска контекст процесу, який активізується.

Основними компонентами підсистеми введення-виведення є драйвери, які управляють зовнішніми пристроями, і файлова система. До підсистеми введення-виведення можна також з деякою долею умовності віднести і диспетчер переривань. Умовність полягає в тому, що диспетчер переривань обслуговує не лише модулі підсистеми введення-виведення, але і інші модулі ОС, зокрема такий важливий модуль, як планувальник/диспетчер потоків.

Файлова система, враховуючи її складність, специфічність і важливість як основного сховища усієї інформації обчислювальної системи заслуговує розгляд в окремому розділі. Проте, тут файлова система розглядається спільно з іншими компонентами підсистеми введення-виведення з двох причин. По-перше, файлова система активно використовує інші частини підсистеми введення-виведення, а по-друге, модель файлу лежить в основі більшості механізмів доступу до пристроїв.

15.1 Фізична організація пристроїв введення-виведення

Облаштування введення-виведення діляться на два типи: блок-орієнтовані пристрої і байт-орієнтовані (потоково-орієнтовані) пристрої.

Блок-орієнтовані пристрої зберігають інформацію в блоках фіксованого розміру, кожен з яких має свою власну адресу, і виконують передачу даних поблочно. Найпоширеніший блок-орієнтований пристрій – диск.

Байт–орієнтовані пристрої не адресуються і не дозволяють робити операцію пошуку, вони генерують або споживають послідовність байтів, тобто виконують передачу даних у вигляді неструктурованих потоків байтів. Прикладами є термінали, мережеві адаптери (комунікаційні порти), клавіатура, маніпулятор «миша». Проте деякі зовнішні пристрої не належать ні до одного класу, наприклад, годинник, який, з одного боку, не адресується, а з іншого боку, не породжує потік байтів. Цей пристрій тільки видає сигнал переривання в деякі моменти часу.

Зовнішній пристрій складається з механічного і електронного компонента. Електронний компонент називається ***контролером пристрою*** або ***адаптером***. Механічний компонент представляє власне пристрій. Деякі контролери можуть управляти декількома пристроями. Якщо інтерфейс між контролером і пристроєм стандартизований, то незалежні виробники можуть випускати сумісні як контролри, так і пристрої.

Операційна система має справу не з пристроєм, а з контролером. Контролер, як правило, виконує прості функції, наприклад, перетворює потік бітів на блоки, які складаються з байтів, і здійснює контроль і виправлення помилок. Кожен контролер має декілька регістрів, які використовуються для взаємодії з центральним процесором. У деяких комп'ютерах ці регістри є частиною фізичного адресного простору. У таких комп'ютерах немає спеціальних операцій уведення-виведення. В інших комп'ютерах адреси регістрів уведення-виведення, які називаються часто портами, утворюють власний адресний простір за рахунок уведення спеціальних операцій уведення-виведення.

ОС виконує введення-виведення, записуючи команди в регістри контролера. Наприклад, контролер гнучкого диска IBM PC приймає 15 команд, таких як READ, WRITE, SEEK, FORMAT тощо. Коли команда прийнята, процесор залишає контролер і займається іншою роботою. При завершенні команди контролер організовує переривання для того, щоб передати управління процесором ОС, яка повинна перевірити результати операції. Процесор отримує результати і статус пристрою, читаючи інформацію з регістрів контролера.

15.2 Задачі ОС з управління файлами і пристроями

Підсистема введення-виведення мультипрограмної ОС при обміні даними із зовнішніми облаштуваннями комп'ютера повинна розв'язувати ряд загальних задач, з яких найважливішими є такі:

- організація паралельної роботи пристроїв введення-виведення і процесора;
- узгодження швидкостей обміну і кешування даних;
- розподіл пристроїв і даних між процесами;
- забезпечення зручного логічного інтерфейсу між пристроями і іншою частиною системи;
- підтримка широкого спектру драйверів з можливістю простого включення в систему нового драйвера;
- динамічне завантаження і вивантаження драйверів;
- підтримка декількох файлових систем;
- підтримка синхронних і асинхронних операцій уведення-виведення.

У наступних підрозділах усі ці задачі розглянемо детальніше.

15.2.1 Організація паралельної роботи пристроїв уведення-виведення

Кожний пристрій уведення-виведення обчислювальної системи – диск, принтер, термінал тощо – забезпечений спеціалізованим блоком управління, який називається контролером. Контролер взаємодіє з драйвером – системним програмним модулем, призначеним для управління цим пристроєм (рис. 15.1). Контролер періодично приймає від драйвера інформацію, яка виводиться на пристрій, а також команди управління, які говорять про те, що з цією інформацією потрібно зробити. Наприклад, вивести у вигляді тексту в певну область терміналу або записати в певний сектор диска.

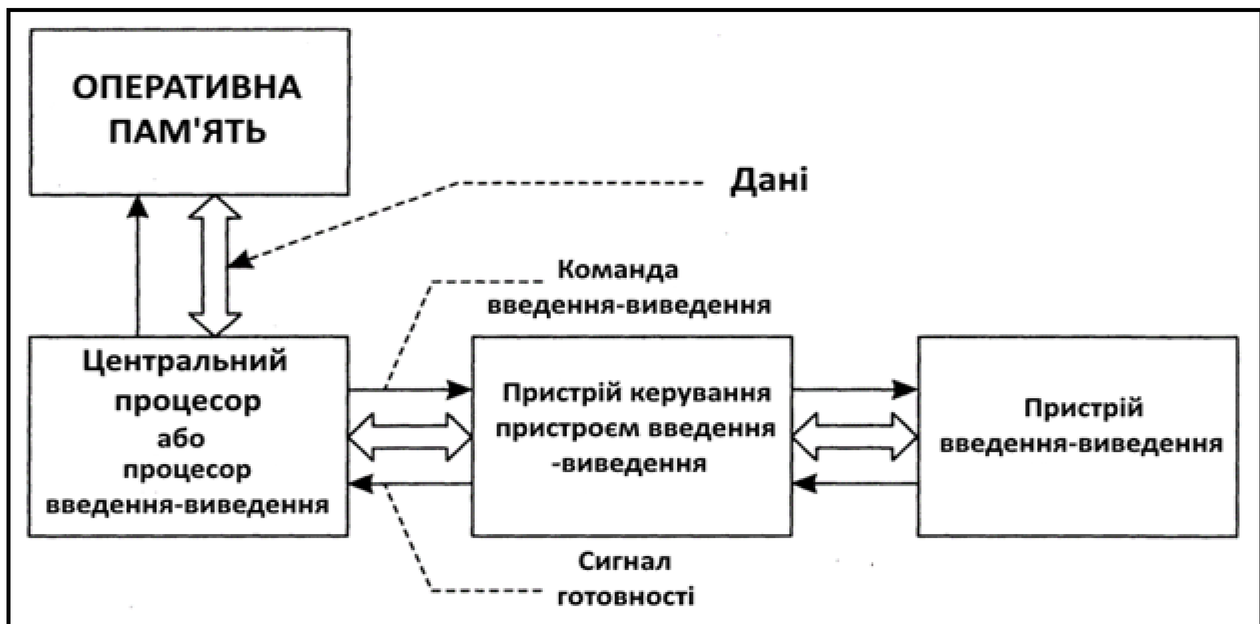


Рисунок 15.1 – Управління введенням-виведенням

Під управлінням контролера пристрій може деякий час виконувати свої операції автономно, не вимагаючи уваги з боку центрального процесора. Цей час залежить від багатьох чинників – об'єму інформації, яка виводиться, міри інтелектуальності контролера, швидкодії пристрою тощо. Навіть найпримітивніший контролер, який виконує прості функції, зазвичай витрачає досить багато час на самостійну реалізацію подібної функції після отримання чергової команди від процесора. Це ж справедливо і для складних контролерів.

Процеси, які відбуваються в контролерах, протікають в періоди між видачами команд незалежно від ОС. Від підсистеми введення-виведення вимагається спланувати в реальному масштабі часу запуск і призупинення великої кількості різноманітних драйверів, забезпечивши прийнятний час реакції кожного драйвера на незалежні події контролера. З іншого боку, необхідно мінімізувати завантаження процесора задачами введення-виведення, залишивши якомога більше процесорного часу на виконання потоків користувача.

Це задача є класичною задачею планування систем реального часу і розв'язується на основі багаторівневої пріоритетної схеми обслуговування за перериваннями. Для забезпечення прийнятного рівня реакції усі драйвери (чи частини драйверів) розподіляються за декількома пріоритетними рівнями відповідно до вимог часу реакції і часу використання процесора.

15.2.2 Узгодження швидкостей обміну і кешування даних

При обміні даними завжди виникає задача узгодження швидкості. Наприклад, якщо один процес користувача обчислює деякі дані і передає їх іншому процесу користувача через оперативну пам'ять, то в загальному випадку швидкості генерації даних і їх читання не співпадають. Узгодження швидкості досягається за рахунок ***буферизації даних*** в оперативній пам'яті і синхронізації доступу процесів до буфера.

У тих спеціалізованих ОС, в яких забезпечення високої швидкості введення-виведення є першочерговою задачею (управління в реальному часі, послуги мережевої файлової служби тощо), велика частина оперативної пам'яті відводиться не під коди прикладних програм, а під буферизацію даних.

Проте буферизація тільки на основі оперативної пам'яті в підсистемі введення-виведення є недостатньою. Різниця між швидкістю обміну з оперативною пам'яттю, куди процеси поміщають дані для обробки, і швидкістю роботи зовнішнього пристрою часто стає занадто великою, щоб в якості тимчасового буфера можна було б використати оперативну пам'ять – її об'єму може просто не вистачити. Для таких випадків необхідно передбачити особливі заходи, і часто як буфер використовується дисковий файл, який називається також ***спул-файлом*** (від spool – шпулька, буфер). Типовий приклад застосування спулінгу дає організація виведення даних на принтер. Для друкарських документів об'єм в декілька десятків мегабайт – не рідкість, тому для їх тимчасового зберігання об'єму оперативної пам'яті явно недостатньо.

Іншим рішенням цієї проблеми є використання великої буферної пам'яті в контролерах зовнішніх пристроїв. Такий підхід особливо корисний в тих випадках, коли переміщення даних на диск занадто уповільнює обмін. Наприклад, в контролерах графічних дисплеїв застосовується буферна пам'ять, більша за об'ємом оперативної пам'яті, і це істотно прискорює виведення графіки на екран.

Буферизація даних дозволяє не лише погоджувати швидкості роботи процесора і зовнішнього пристрою, але і розв'язати іншу задачу – скоротити кількість реальних операцій уведення-виведення за рахунок кешування даних. Дисковий кеш є неодмінним атрибутом підсистем уведення-виведення практично в усіх операційних систем, значно скорочуючи час доступу до даних.

15.2.3 Розподіл пристроїв і даних між процесами

Пристрої уведення-виведення може надаватися процесам як в ***монопольне, так і в загальне (що розділяється)*** використання. При цьому ОС повинна забезпечувати контроль доступу тими ж способами, що і при доступі процесів до інших ресурсів обчислювальної системи – шляхом перевірки прав користувача або групи користувачів, від імені яких діє процес, на виконання тієї або іншої операції над пристроєм. Наприклад, певній групі користувачів послідовний порт дозволено забирати в монопольне володіння, а іншим користувачам це заборонено.

Операційна система може контролювати доступ не лише до пристрою в цілому, але і до окремих областей даних, які зберігаються або відображаються цим пристроєм. Диск є типовим прикладом пристрою, для якого важливо контролювати доступ не до пристрою в цілому, а до окремих каталогів і файлів.

При виведенні інформації на графічний дисплей окремі вікна екрану також є ресурсами, до яких необхідно забезпечити той або інший вид доступу для процесів, які протікають в системі. При цьому для кожної області даних або частини пристрою можуть бути задані свої права доступу, не пов'язані прямо з

правами доступу до пристрою в цілому. Так, у файловій системі для кожного каталогу і файлу можна задати індивідуальні права доступу. Очевидно, що для організації загального доступу до частин пристрою або частин даних, які зберігаються на ньому, неодмінною умовою є задання режиму загального використання пристрою в цілому.

Один і той же пристрій в різні періоди часу може використовуватися як в режимах, які розділяються, так і в монопольних. Проте існують пристрої, для яких характерний один з цих режимів. Наприклад, послідовні порти і алфавітно-цифрові термінали частіше використовуються в монопольному режимі, а диски – в режимі загального доступу. Операційна система повинна надавати ці пристрої в обох режимах, здійснюючи відстежування процедур захоплення і звільнення монопольно використовуваних пристроїв, а у разі загального використання, оптимізуючи послідовність операцій введення-виведення для різних процесів з метою підвищення загальної продуктивності, якщо це можливо.

Наприклад, при обміні даними декількох процесів з диском можна так упорядкувати послідовність операцій, що непродуктивні витрати часу на переміщення головок істотно зменшуються. При цьому для окремих процесів можливе деяке уповільнення операції введення-виведення.

При розподілу пристрою між процесами може виникнути необхідність у розмежуванні області даних двох процесів один від одного. Така потреба виникає при загальному використанні так званих послідовних пристроїв, дані в яких на відміну від пристроїв прямого доступу не адресуються. Типовим представником такого роду пристроїв є принтер, який не виділяється в монопольне володіння процесам, і в той же час кожен документ має бути надрукований у вигляді послідовного набору сторінок. Для подібних пристроїв організовується черга завдань на виведення, при цьому кожне завдання є порцією даних, яку не можна розривати, наприклад, документ для друку. Для зберігання черги завдань використовується спул-файл, який одночасно погоджує швидкості роботи принтера і оперативної пам'яті і дозволяє організувати розбиття даних на логічні порції. Оскільки спул-файл знаходиться на пристроях прямого доступу, то процеси можуть одночасно виконувати виведення на принтер, поміщаючи дані у свій розділ спул-файлу.

15.2.4 Забезпечення логічного інтерфейсу між пристроями

Різноманітність пристроїв введення-виведення робить особливо актуальною функцію ОС зі створення екрануючого логічного інтерфейсу між периферійними пристроями і додатками. Практично всі сучасні операційні системи підтримують в якості основи такого інтерфейсу файлову модель периферійних пристроїв, коли будь-який пристрій виглядає для прикладного програміста послідовним набором байтів. З цим набором байтів можна працювати за допомогою уніфікованих системних викликів (наприклад, read і write), задаючи ім'я файлу-пристрою і зміщення від початку послідовності байтів. Для підтримки такого інтерфейсу підсистема введення-виведення

повинна виконати досить велику роботу, враховуючи різницю в організації операцій обміну даними, наприклад, з жорстким диском і графічним терміналом.

Привабливість моделі файлу-пристрою полягає в її простоті і уніфікованості для обладнання будь-якого типу, проте в багатьох випадках для програмування операцій уведення-виведення деякого пристрою вона є занадто бідною. Тому ця модель часто використовується тільки як базис, над яким підсистема введення-виведення будує змістовнішу модель пристрою конкретного типу. Підсистема введення-виведення надає, як правило, специфічний інтерфейс для виведення графічної інформації на дисплей або принтер, для програмування операцій мережевого обміну тощо. При цьому розробник специфічного інтерфейсу завжди може спиратися на наявний базовий інтерфейс.

15.2.5 Підтримка широкого спектру драйверів

Перевагою підсистеми введення-виведення будь-якої універсальної ОС є наявність різноманітного набору драйверів для найпопулярніших периферійних пристроїв. Прекрасно спланована і реалізована операційна система може потерпіти невдачу на ринку тільки через те, що до її складу не включений достатній набір драйверів. У цьому випадку адміністратори і користувачі змушені шукати потрібний їм драйвер для наявного у них зовнішнього пристрою у виробників устаткування або, що ще гірше, займатися його розробкою. Саме в такій ситуації опинилися користувачі перших версій OS/2, і, можливо, ця обставина послужила свого часу не останньою причиною здачі позицій цієї непоганої операційної системи багатій на драйвери ОС Windows 3.x.

Щоб операційна система не відчувала нестачі в драйверах, потрібна наявність чіткого, зручного і відкритого інтерфейсу між драйверами і іншими компонентами ОС. Такий інтерфейс потрібний для того, щоб драйвери писали не лише безпосередні розробники цієї операційної системи, але і велика армія програмістів по всьому світу, в першу чергу – тих підприємств, які випускають зовнішні пристрої для комп'ютерів. Відкритість інтерфейсу драйверів, тобто доступність його опису для незалежних розробників програмного забезпечення, є необхідною умовою успішного розвитку операційної системи.

Драйвер взаємодіє, з одного боку, з модулями ядра ОС (модулями підсистеми введення-виведення, модулями системних викликів, модулями підсистем управління процесами і пам'яттю), а з іншого боку – з контролерами зовнішніх пристроїв (рис. 15.2). Тому існують два типи інтерфейсів: ***інтерфейс «драйвер-ядро»*** (Driver Kernel Interface, DKI) і інтерфейс ***«драйвер-пристрій»*** (Driver Device Interface, DDF).

Інтерфейс «драйвер-ядро» має бути стандартизованим у будь-якому випадку. Іінтерфейс «драйвер-пристрій» має сенс стандартизувати тоді, коли підсистема введення-виведення не дозволяє драйверу безпосередньо взаємодіяти з апаратурою контролера, а виконує ці операції самостійно. Екранування драйвера від апаратури є дуже корисною функцією, оскільки драйвер в цьому випадку стає незалежним від апаратної платформи.

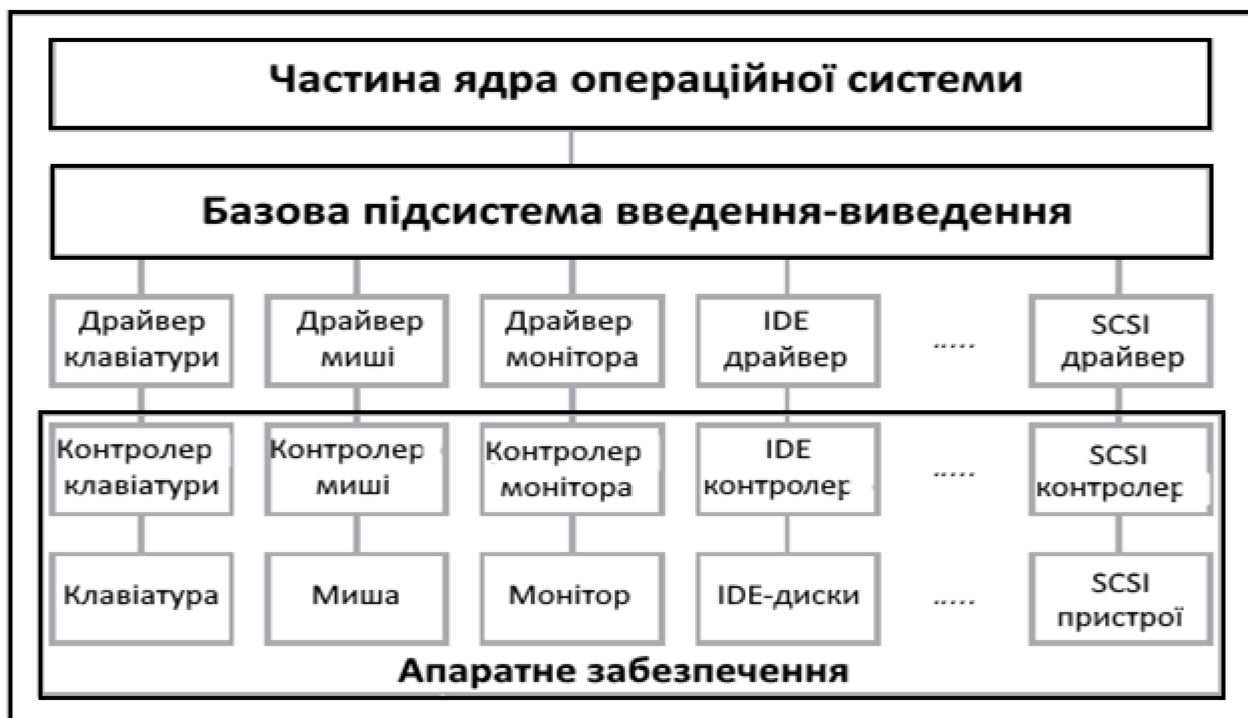


Рисунок 15.2 – Структура системи введення-виведення

Підсистема введення-виведення може підтримувати декілька різних типів інтерфейсів DKI/DDI, надаючи специфічний інтерфейс для пристроїв певного класу.

Так, в ОС Windows NT для драйверів мережевих адаптерів існує інтерфейс стандарту NDIS (Network Driver Interface Specification), тоді як драйвери мережевих транспортних протоколів взаємодіють з верхніми шарами мережевого програмного забезпечення по інтерфейсу TDI (Transport Driver Interface).

Підсистема введення-виведення підтримує велику кількість системних функцій, які драйвер може викликати для виконання деяких типових дій. Прикладами можуть служити згадані операції обміну з регістрами контролера, ведення буферів для проміжного зберігання даних введення-виведення, синхронізація роботи декількох драйверів, копіювання даних з призначеного для користувача простору в простір системи тощо.

Для підтримки процесу розробки драйверів операційної системи випускається так званий пакет DDK (Driver Development Kit) – набір відповідних інструментальних засобів: бібліотек, компіляторів і відладчиків.

15.2.6 Динамічне завантаження і вивантаження драйверів

Окрім проблеми розробки нових драйверів існує також проблема включення драйвера до складу модулів працюючої ОС, тобто динамічного завантаження-вивантаження драйвера. Оскільки набір потенційно підтримуваних цією ОС периферійних пристроїв завжди істотно ширше набору пристроїв, якими ОС повинна управляти при установці на конкретній машині, то цінною властивістю ОС є можливість динамічно завантажувати в оперативну

пам'ять необхідний драйвер (без зупинки ОС), і вивантажувати його після того, як потреба в підтримці пристрою минула, що може істотно заощадити системну область пам'яті.

Альтернативою динамічному завантаженню драйверів при зміні поточної конфігурації зовнішніх пристроїв комп'ютера є повторна компіляція коду ядра з необхідним набором драйверів, який створює між усіма компонентами ядра статичні зв'язки замість динамічних. Наприклад, таким чином вирішувалася ця проблема в ранніх версіях операційної системи UNIX. При статичних зв'язках між ядром і драйверами структура ОС спрощується, але цей підхід вимагає наявності початкових кодів модулів операційної системи, доступність яких швидше є виключенням, а не правилом. Крім того, в цьому варіанті працюючу попередню версію операційної системи необхідно зупинити і замінити новою, а перерви в роботі ОС в деяких додатках можуть і не допускатися.

Підтримка динамічного завантаження драйверів є практично обов'язковою вимогою для сучасних універсальних операційних систем.

15.2.7 Підтримка декількох файлових систем

Диски представляють особливий рід периферійних пристроїв, оскільки саме на них зберігається велика частина як призначених для користувача, так і системних даних. Дані на дисках організовуються у файлові системи, і властивості файлової системи багато в чому визначають властивості самої ОС – її відмовостійкість, швидкодію, максимальний об'єм даних.

Популярність файлової системи часто призводить до її міграції з «рідної» ОС в інші операційні системи. Наприклад, файлова система FAT з'явилася спочатку у MS-DOS, але потім була реалізована в OS/2, сімействі MS Windows і багатьох реалізаціях UNIX. Зважаючи на це підтримка декількох популярних файлових систем для підсистеми введення-виведення також важлива, як і підтримка широкого спектру периферійних пристроїв. Важливо також, щоб архітектура підсистеми введення-виведення дозволяла досить просто включати до її складу нові типи файлових систем, без необхідності переписування коду.

15.2.8 Синхронні і асинхронні операції введення-виведення

Операція введення-виведення може виконуватися стосовно програмного модуля, який запросив операцію, в ***синхронному*** або ***асинхронному*** режимах. Сенс цих режимів той же, що і для розглянутих вище системних викликів. Синхронний режим означає, що програмний модуль призупиняє свою роботу до тих пір, поки операція введення-виведення не буде завершена (рис. 15.2, *а),* а при асинхронному режимі програмний модуль продовжує виконуватися одночасно з операцією введення-виведення (рис. 15.2*, б).*

Відмінність же полягає в тому, що операція введення-виведення може бути ініційована не лише призначеним для користувача процесом – в цьому випадку операція виконується в рамках системного виклику, але і кодом ядра, наприклад кодом підсистеми віртуальної пам'яті для прочитання відсутньої в пам'яті сторінки.

Підсистема введення-виведення повинна надавати своїм клієнтам (процесам користувача і ядра) можливість виконувати як синхронні, так і асинхронні операції введення-виведення, залежно від потреб процесу. Системні виклики введення-виведення частіше оформляються як синхронні процедури в зв'язку з тим, що такі операції тривають довго і процесу користувача або потоку все одно доведеться чекати отримання результатів операції для того, щоб продовжити свою роботу.

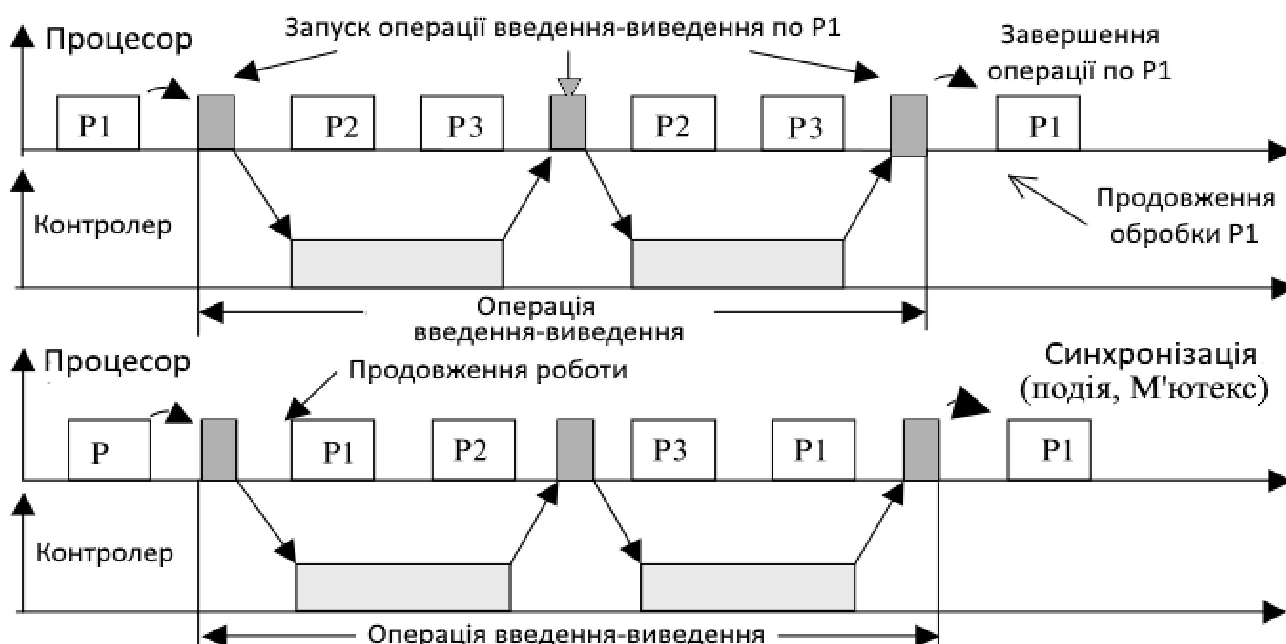


Рисунок 15.2 – Два режими виконання операцій введення-виведення

Внутрішні ж виклики операцій уведення-виведення з модулів ядра зазвичай виконуються у вигляді асинхронних процедур, оскільки кодам ядра потрібна свобода у виборі подальшої поведінки після запиту операції введення-виведення.

15.3 Організація програмного забезпечення введення-виведення

Основна ідея організації програмного забезпечення введення-виведення полягає в розбитті його на декілька рівнів (шарів), причому нижні рівні забезпечують екранування особливостей апаратури від верхніх, а ті, у свою чергу, забезпечують зручний інтерфейс для користувачів.

Ключовим принципом є незалежність від пристроїв. Вид програми не повинен залежати від того, читає вона дані з гнучкого чи з жорсткого диска. Дуже близькій до ідеї незалежності від пристроїв є ідея однакового іменування, тобто для іменування пристроїв мають бути прийняті єдині правила.

Іншим важливим питанням для програмного забезпечення введення-виведення є обробка помилок. Помилки слід обробляти як можна ближче до апаратури. Якщо контролер виявляє помилку читання, то він повинен спробувати її скоректувати. Якщо ж це йому не вдається, то виправленням помилок повинен зайнятися драйвер пристрою. Багато помилок можуть зникати

при повторних спробах виконання операцій введення-виведення. наприклад, помилки, викликані наявністю порошинок на головках читання або на диску. І тільки якщо нижній рівень не може впоратися з помилкою, він повідомляє про помилку верхній рівень.

Ще одне ключове питання – це використання ***блокуючих (синхронних)*** і ***неблокуючих (асинхронних) передач***. Більшість операцій фізичного введення-виведення виконуються асинхронно – процесор починає передачу і переходить на іншу роботу, поки не настає переривання. Програми користувача набагато легше писати, якщо операції введення-виведення блокуючі . ОС виконує операції введення-виведення асинхронно, але представляє їх для програм користувача в синхронній формі. Остання проблема полягає в тому, що одні пристрої є такими, що ***розділяються***, а інші **–** ***виділеними***. Диски – це пристрої, які розділяються, оскільки одночасний доступ декількох користувачів до диска не є проблемою. Принтери – це виділені пристрої, тому що не можна змішувати рядки, які друкуються різними користувачами. Наявність виділених пристроїв створює для операційної системи деякі проблеми.

Для вирішення поставлених проблем доцільно розділити програмне забезпечення введення-виведення на чотири шари (рис. 15.3):

1) обробка переривань;
2) драйвери пристроїв;
3) незалежний від пристроїв шар операційної системи;
4) призначений для користувача шар програмного забезпечення.

Рисунок 15.3 – Багаторівнева організація підсистеми введення-виведення

Багатошарова побудова програмного забезпечення, характерна для операційних систем, виявляється особливо природною і корисною при побудові підсистеми введення-виведення при великій різноманітності пристроїв введення-виведення, які мають істотно різні характеристики (принтер і диски, графічний монітор і мережевий адаптер тощо).

Ієрархічна структура програмного забезпечення дозволяє дотримуватися балансу між двома дуже суперечливими вимогами. З одного боку, необхідно врахувати всі особливості кожного пристрою, а з іншого – забезпечити єдине логічне представлення і уніфікований інтерфейс для пристроїв усіх типів. При цьому нижні шари підсистеми введення-виведення повинні включати індивідуальні драйвери, написані для конкретних фізичних пристроїв, а верхні шари повинні узагальнювати процедури управління цими пристроями.

Також можна надавати спільний інтерфейс, якщо не для всіх пристроїв, то принаймні для груп пристроїв, які мають деякі загальні характеристики, наприклад для принтерів певного виробника або для всіх матричних принтерів.

Обробка переривань. Переривання мають бути приховані як можна глибше в надрах операційної системи, щоб як можна менша частина ОС мала з ними справу. Найкращий спосіб полягає в дозволі процесу, який ініціював операцію введення-виведення, блокувати себе до завершення операції і настання переривання. Процес може блокувати себе, використовуючи, наприклад, виклик DOWN для семафора, або виклик WAIT для змінної умови, або виклик RECEIVE для очікування повідомлення. При настанні переривання процедура обробки переривання виконує розблокування процесу, який ініціював операцію введення-виведення, використовуючи виклики UP, SIGNAL або посилаючи процесу повідомлення. У будь-якому випадку ефект від переривання полягатиме в тому, що раніше заблокований процес тепер продовжить своє виконання.

Драйвери пристроїв. Увесь залежний від пристрою код поміщається в драйвер пристрою. Кожен драйвер управляє пристроями одного типу або, можливо, одного класу.

У операційній системі тільки драйвер пристрою знає про конкретні особливості якого-небудь пристрою. Наприклад, тільки драйвер диска має справу з доріжками, секторами, циліндрами, часом встановлення головки і іншими чинниками, які забезпечують правильну роботу диска.

Драйвер пристрою приймає запит від пристрою програмного шару і вирішує, як його виконати. Типовим запитом є читання *n* блоків даних. Якщо драйвер був вільний під час надходження запиту, то він починає виконувати запит негайно. Якщо ж він був зайнятий обслуговуванням іншого запиту, то запит, який знову поступив, приєднується до черги інших запитів, і він буде виконаний, коли настане його черга.

Перший крок в реалізації запиту введення-виведення, наприклад, для диска, полягає в перетворенні його з абстрактної форми в конкретну. Для дискового драйвера це означає перетворення номерів блоків на номери циліндрів, головок, секторів, перевірку того, чи працює мотор чи знаходиться головка над потрібним циліндром. Коротше кажучи, він повинен вирішити, які операції контролера потрібно виконати і в якій послідовності.

Після передачі команди контролеру драйвер повинен вирішити, чи блокувати себе до закінчення заданої операції чи ні. Якщо операція займає значний час, як при друці деякого блоку даних, то драйвер блокується до тих пір, поки операція не завершиться, і обробник переривання не розблокує його. Якщо команда введення-виведення виконується швидко (наприклад, прокрутка екрану), то драйвер чекає її завершення без блокування.

Незалежний від пристроїв шар операційної системи. Велика частина програмного забезпечення введення-виведення є незалежною від пристроїв. Точна межа між драйверами і незалежними від пристроїв програмами визначається системою, оскільки деякі функції, які могли б бути реалізовані незалежним способом, насправді виконані у вигляді драйверів для підвищення ефективності або з інших причин.

Типовими функціями для незалежного від пристроїв шару є:

- забезпечення загального інтерфейсу до драйверів пристроїв;
- іменування пристроїв;
- захист пристроїв;
- забезпечення незалежного розміру блоку;
- буферизація;
- розподіл пам'яті на блок-орієнтованих пристроях;
- розподіл і звільнення виділених пристроїв;
- повідомлення про помилки.

Зупинимося на деяких функціях цього переліку. Верхнім шарам програмного забезпечення незручно працювати з блоками різної величини, тому цей шар забезпечує єдиний розмір блоку, наприклад, за рахунок об'єднання декількох різних блоків в єдиний логічний блок. У зв'язку з цим верхні рівні мають справу з абстрактними пристроями, які використовують єдиний розмір логічного блоку незалежно від розміру фізичного сектора. При створенні файлу або заповненні його новими даними необхідно виділити йому нові блоки. Для цього ОС повинна вести список або бітову карту вільних блоків диска. На підставі інформації про наявність вільного місця на диску може бути розроблений алгоритм пошуку вільного блоку, який незалежний від пристрою і реалізовується програмним шаром, що знаходиться вище за шар драйверів.

Призначений для користувача шар програмного забезпечення. Хоча велика частина програмного забезпечення введення-виведення знаходиться всередині ОС, деяка його частина міститься в бібліотеках, які зв'язуються з призначеними для користувача програмами. Системні виклики, які включають виклики введення-виведення, робляться бібліотечними процедурами. Якщо програма, написана на мові *C,* містить виклик *count = write(fd, buffer, nbytes),* то бібліотечна процедура *write* буде пов'язана з програмою.

Набір подібних процедур є частиною системи введення-виведення. Зокрема, форматування введення або виведення виконується бібліотечними процедурами. Прикладом може служити функція *printf* мови *C*, яка приймає рядок формату і, можливо, деякі змінні, як вхідну інформацію, потім будує рядок символів ASCII і робить виклик *write* для виведення цього рядка. Стандартна

бібліотека введення-виведення містить велике число процедур, які виконують уведення-виведення і працюють як частина програми користувача.

Іншою категорією програмного забезпечення введення-виведення є підсистема ***спулінгу*** (spooling). Спулінг – це спосіб роботи з виділеними пристроями в мультипрограмній системі. Розглянемо типовий пристрій, який вимагає спулингу – принтер. Хоча технічно легко дозволити кожному процесу користувача відкрити спеціальний файл, пов'язаний з принтером, такий спосіб небезпечний через те, що процес користувача може монополізувати принтер на довільний час. Замість цього створюється спеціальний процес – монітор, який отримує виняткові права на використання цього пристрою. Також створюється спеціальний каталог, який називається каталогом спулінгу. Для того щоб надрукувати файл, процес користувача поміщає інформацію, яка виводиться, в цей файл, і поміщає його в каталог спулінгу. Процес-монітор по черзі роздруковує всі файли, які містяться в каталозі спулінгу.

15.4 Буферизація операцій введення-виведення

Припустимо, що процесу користувача необхідно виконати зчитування блоків даних завдовжки 512 байт кожний по одному з гнучкого диска. Дані будуть розміщені в область всередині адресного простору процесу користувача з віртуальною адресою від 1000 до 1511. Найпростіший шлях розв'язання цієї задачі – виконати команду введення-виведення і очікувати дані.

При генерації процесом команди введення-виведення він призупиняється і вивантажується на диск до початку її виконання. Далі процес чекає, коли буде виконана запрошена ним операція введення-виведення, яка, у свою чергу, чекає, коли процес буде повернений в основну пам'ять, оскільки місце в основній пам'яті для зчитування даних просто відсутнє. Для того щоб уникнути взаємоблокування, призначена для користувача пам'ять в операціях введення-виведення має бути заблокована в основній пам'яті відразу ж після видачі запиту на введення-виведення.

Те ж міркування застосовне і до операції виведення. Щоб зменшити накладні витрати і збільшити ефективність, іноді зручно виконувати читання даних заздалегідь, до реального запиту, а запис даних – трохи пізніше за реальний запит. Ця методика відома як ***буферизація.*** Розглянемо деякі схеми буферизації, підтримувані ОС для підвищення продуктивності (рис. 15.4).

Введення без буферу. При небуферованому введенні (рис. 15.4, *а)* пересилка здійснюється посимвольно. Після кожного символу відбувається переривання.

Одинарний буфер. Простим типом підтримки з боку ОС є одинарний буфер. У той момент, коли процес користувача виконує запит уведення-виведення, ОС призначає йому буфер в системній частині основної пам'яті (рис. 15.4, *b*).

Схема одинарного буфера для блочно-орієнтованих пристроїв може бути описана таким чином.

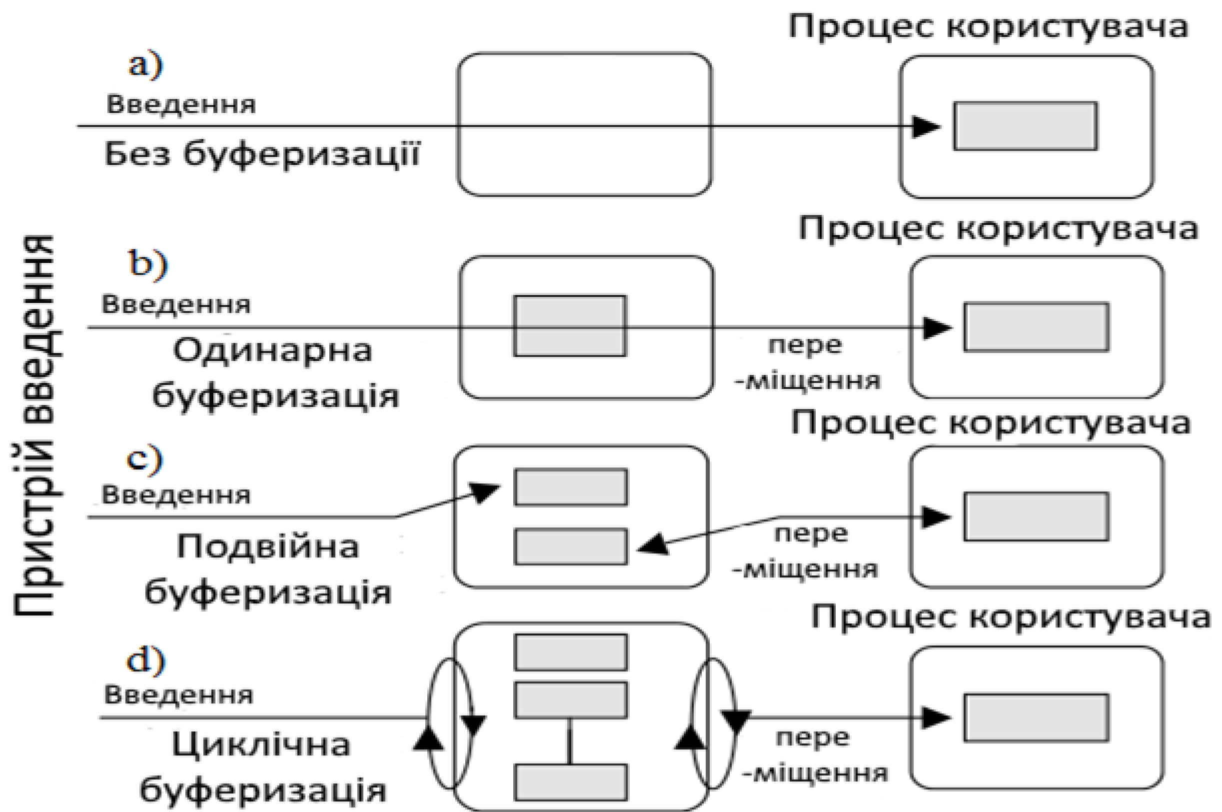


Рисунок 15.4 – Схеми буферизації введення-виведення

Спочатку здійснюється передача вхідних даних у системний буфер. Коли вона завершується, процес переміщає блок у простір користувача і негайно робить запит наступного блоку. Така процедура називається ***випереджаючим прочитанням,*** або ***застережливим введенням***. Вона виконується в припущенні, що цей блок з часом знадобиться.

Такий підхід, у порівнянні з відсутністю буферизації, забезпечує підвищення швидкодії. Призначений для користувача процес може обробляти один блок даних в той час, коли відбувається прочитання наступного блоку. ОС при цьому може здійснювати вивантаження процесу, оскільки виконується операція зчитування даних в системну область, а не в пам'ять процесу користувача.

Проте така технологія ускладнює функціонування ОС, яка повинна стежити за виділенням системних буферів. Впливає буферизація і на схему підкачування. Коли операція введення-виведення працює з тим же диском, який використовується і для свопінгу, втрачається сенс в організації черги операцій запису. Вивантаження процесу і звільнення основної пам'яті не почнеться до тих пір, поки не завершиться запрошена операція введення-виведення – а тоді вивантаження процесу не матиме сенсу.

Подвійна буферизація в ядрі. Удосконалити схему одинарної буферизації можна шляхом використання двох системних буферів (рис. 15.4, *с)*. Тепер процес виконує передачу даних в один буфер (чи зчитування з нього), тоді як ОС звільняє (чи заповнює) інший. Ця технологія відома як ***подвійна буферизація*** або ***змінний буфер***.

Циклічний буфер. Схема подвійного буфера покликана вирівняти потік даних між пристроями введення-виведення і процесом. Якщо нас цікавить продуктивність деякого процесу, то в першу чергу вимагається, щоб операції введення-виведення не гальмували його роботи. Подвійна буферизація може виявитися недостатньою, якщо процес часто виконує введення або виведення. Частенько в такому разі розв'язати проблему допомагає нарощування буферів.

При використанні великої кількості буферів, яка складається більше ніж з двох, схема іменується ***циклічною буферизацією*** (рис. 15.4, *d*).

Проте при такій роботі запис або читання великої кількості інформації з адресного простору введення-виведення призводять до великої кількості операцій введення-виведення, які повинен виконувати процесор. Для звільнення процесора від операцій послідовного виведення даних з оперативної пам'яті або послідовного введення в неї був запропонований механізм прямого доступу зовнішніх пристроїв до пам'яті – ПДП або Direct Memory Access – **DMA**.

Контролер містить декілька регістрів: регістр адреси пам'яті, лічильник байтів і управляючі регістри (порт введення-виведення, читання або запис).

Для того щоб будь-який пристрій, окрім процесора, міг записати інформацію в пам'ять або прочитати її з пам'яті, необхідно, щоб цей пристрій міг забрати у процесора управління локальною магістраллю для виставляння відповідних сигналів на шини адреси, даних і управління. Для централізації ці обов'язки покладаються не на кожен пристрій окремо, а на спеціальний контролер – контролер прямого доступу до пам'яті.

Контролер прямого доступу до пам'яті має декілька спарених ліній – каналів DMA, які можуть підключатися до різних пристроїв. Перед початком використання прямого доступу до пам'яті цей контролер необхідно **запрограмувати**, записавши в його порти інформацію про те, який канал або канали передбачається задіяти, які операції вони здійснюватимуть, яка адреса пам'яті є початковою для передачі інформації і яка кількість інформації має бути передана.

Отримавши по одній з ліній (каналів DMA), сигнал запиту на передачу даних від зовнішнього пристрою, контролер по шині управління повідомляє процесор про бажання взяти на себе управління локальною магістраллю. Процесор, можливо, через деякий час, необхідний для завершення його дій з магістраллю, передає управління нею контролеру DMA, сповістивши його спеціальним сигналом. Контролер DMA виставляє на адресну шину адреси пам'яті для передачі чергової порції інформації і другою лінією каналу прямого доступу до пам'яті повідомляє пристрій про готовність магістралі до передачі даних.

Після цього, використовуючи шину даних і шину управління, контролер DMA, пристрій введення-виведення і пам'ять здійснюють процес обміну інформацією. Потім контролер прямого доступу до пам'яті сповіщає процесор про свою відмову від управління магістраллю, і той берет керівні функції на себе. При передачі великої кількості даних увесь процес повторюється циклічнно.

Роботу з контролером DMA можна простежити за схемою, показаною на рис.15.5:

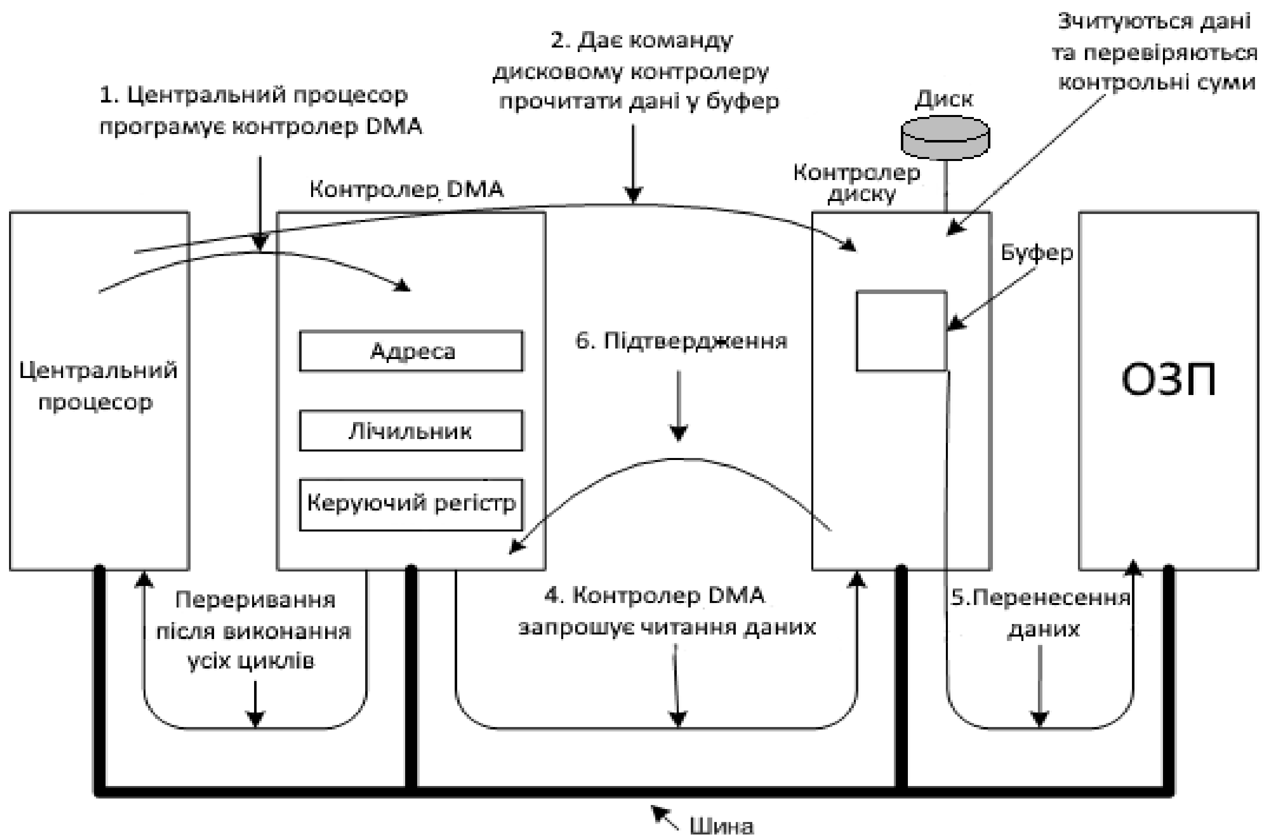


Рисунок 15.5 – Робота DMA-контролера

1. Процесор програмує контролер (які дані і куди перемістити).
2. Процесор дає команду дисковому контролеру прочитати дані в буфер.
3. Зчитуються дані в буфер, контролер диска перевіряє їх контрольну суму.
4. Контролер DMA посилає запит на читання дисковому контролеру.
5. Контролер диску поставляє дані на шину, адрес пам'яті вже знаходиться на шині, відбувається запис даних в пам'ять.
6. Коли запис закінчений, контролер диска посилає підтвердження DMA контролеру.
7. DMA контролер збільшує використовувану адресу і зменшує значення лічильника байтів.
8. Все повторюється з пункту 4, поки значення лічильника не стане рівним нулю.
9. Контролер DMA ініціює переривання.

При прямому доступі до пам'яті процесор і контролер DMA по черзі управляють локальною магістраллю. Це, звичайно, дещо знижує продуктивність процесора, оскільки при виконанні деяких команд або при читанні чергової порції команд у внутрішній кеш він повинен чекати звільнення магістралі, але в цілому продуктивність обчислювальної системи істотно росте.

При підключенні до системи нового пристрою, який уміє використовувати прямий доступ до пам'яті, необхідно програмно або апаратно задати номер каналу DMA, до якого буде приписано пристрій. На відміну від переривань, де один номер переривання міг відповідати декільком пристроям, канали DMA завжди перебувають у монопольному володінні пристроями.

15.6 Дискове планування

З кожним роком збільшення швидкості процесорів і об'єму основної пам'яті здійснюється з великим випередженням у порівнянні зі швидкістю доступу до диска. В результаті швидкість звернення до дисків зараз, щонайменше на чотири порядки менше швидкості звернення до основної пам'яті, і цей розрив, схоже, в майбутньому тільки збільшуватиметься. Тому продуктивність дискової системи є життєво важливим питанням, і величезна кількість дослідницьких робіт спрямована на пошук схем її удосконалення. Тут ми розглянемо деякі ключові питання і найважливіші підходи в цій області.

Перш ніж приступити до безпосереднього викладу самих алгоритмів, згадаємо внутрішній устрій жорсткого диска і визначимо, які параметри запитів ми можемо використати для планування (диспетчеризації).

15.6.1 Будова жорсткого диска і параметри планування

Сучасний жорсткий магнітний диск є набором круглих пластин, які знаходяться на одній осі, покритих з однієї або двох сторін спеціальним магнітним шаром (рис. 15.6). Біля кожної робочої поверхні кожної пластини розташовані магнітні головки для читання і запису інформації. Ці головки приєднані до спеціального важеля, який може переміщати увесь блок головок над поверхнями пластин як єдине ціле.

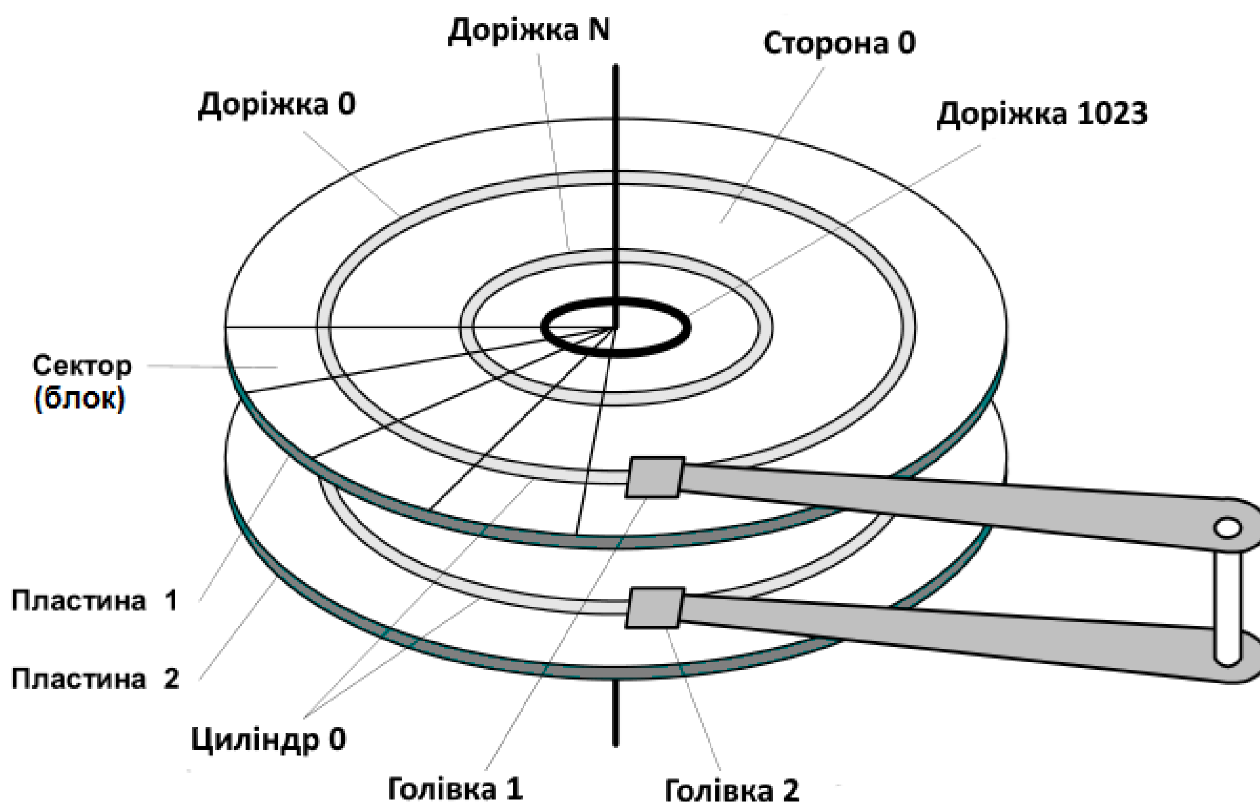


Рисунок 15.6 – Схема облаштування жорсткого диска

Поверхні пластин розділені на концентричні кола, всередині яких, власне, і може зберігатися інформація. Набір концентричних кіл на усіх пластинах для

одного положення головок утворює ***циліндр.*** Кожне кільце всередині циліндра одержало назву ***доріжки.*** Усі доріжки діляться на рівне число ***секторів***. Кількість доріжок, циліндрів і секторів може варіюватися від одного жорсткого диска до іншого. Як правило, сектор є мінімальним об'ємом інформації, яка може бути прочитана з диска за один раз.

При роботі диска набір пластин обертається навколо своєї осі з високою швидкістю, підставляючи по черзі під головки відповідних доріжок усі їх сектори. Номер сектора, номер доріжки і номер циліндра однозначно визначають положення даних на жорсткому диску і, разом з типом здійснюваної операції – читання або запис – повністю характеризують частину запиту, пов'язану з пристроєм, при обміні інформацією в об'ємі одного сектора.

15.6.2 Параметри продуктивності диска

Конкретні деталі дискової операції введення-виведення залежать від комп'ютерної системи, операційної системи, природи каналу введення-виведення і апаратного забезпечення контролера диска (рис. 15.7) [12].

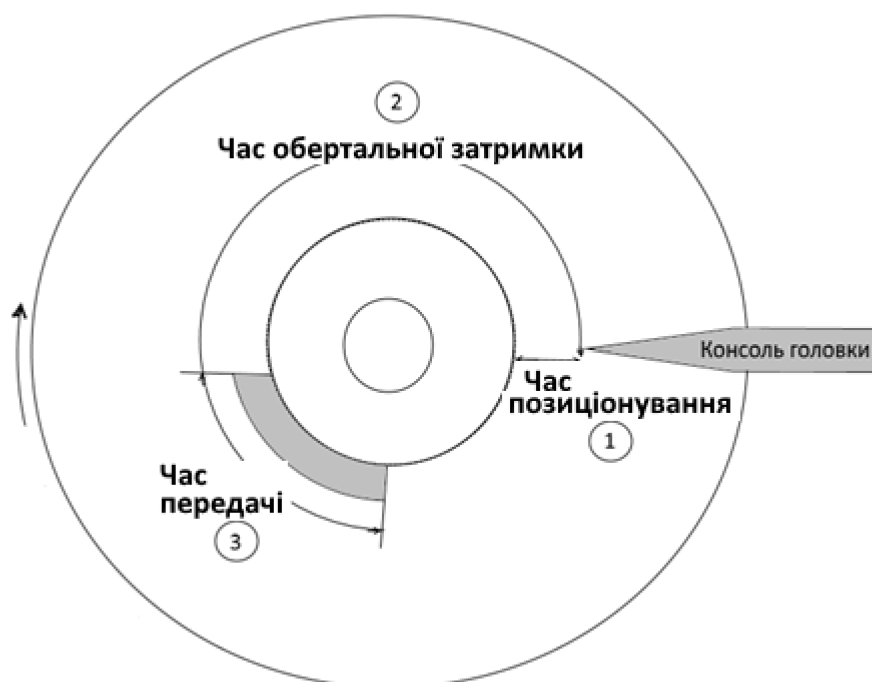


Рисунок 15.7 – Складові часу дискового доступу

При роботі диска його швидкість обертання постійна. Для того щоб виконати читання або запис, головка повинна знаходитися над шуканою доріжкою, і, крім того, – над початком шуканого сектора на цій доріжці. Процедура вибору доріжки включає переміщення головки (у системі з рухливими головками) або електронний вибір потрібної головки (у системі з нерухомими головками) до потрібного циліндра (тобто виконати **операцію позиціонування).** У системі з рухливими головками на позиціонування головки над доріжкою витрачається час, відомий як ***час позиціонування.*** У будь-якому випадку після вибору доріжки контролер диска чекає на момент, коли початок шуканого сектора досягне головки. Час, необхідний для досягнення головки з

початком сектора, відомий як ***час затримки із-за обертання***, або ***час очікування обертання***. Потім запис повинна повністю пройти під головкою в ході обертання диска. Час проходження називається ***часом передачі***.

Сума часу пошуку і часу затримки із-за обертання складає ***час доступу*** – час, який потрібний для позиціонування для читання або запису. Як тільки головка потрапляє у вказану позицію, виконується операція читання або запису, здійснювана під час руху сектора під головкою, – це і є безпосередня передача даних при виконанні операції введення-виведення.

Окрім цього існує ряд інших затримок. Коли процес виконує запит на введення-виведення, останній має бути розміщений в черзі пристрою. Після цього виконується призначення пристрою процесу. Якщо пристрій використовує канали введення-виведення спільно з іншими дисками, потрібне додаткове очікування доступності каналу. І тільки після цього здійснюється безпосередній доступ до диска, розглянутий раніше.

15.6.3 Стратегії дискового планування

Розглянемо типову ситуацію в багатозадачному середовищі, коли ОС підтримує чергу запитів для кожного пристрою введення-виведення. Відповідно, в черзі одного диска знаходитиметься деяка кількість запитів на введення-виведення від різних поцесів. Якщо вибирати запити з черги випадковим чином, то слід чекати, що шукані доріжки розташовуватимуться в довільному порядку, який призведе до дуже низької продуктивності. Такий випадковий розподіл може служити точкою відліку для оцінки інших методик [22].

Оскільки час позиціонування більше, ніж затримка внаслідок обертання диска, більшість алгоритмів диспетчеризації (планування) концентруються на мінімізації загального часу позиціонування для набору запитів. При зменшенні різниці між часом позиціонування і затримкою внаслідок обертання мінімізація останньої теж може істотно поліпшити продуктивність, особливо при великих навантаженнях на накопичувач.

15.7 Диспетчеризація дискових операцій

Алгоритми диспетчеризації дискових операцій, які використовуються системою, залежать від призначення цієї системи, але більшість алгоритмів оцінюються за такими загальними критеріями [12]:

1. **Пропускна спроможність** – кількість запитів, що виконуються за одиницю часу.
2. **Середній час реагування** – середній час, що проходить між надходженням запиту і його виконанням.
3. **Розкид часу реагування** – рівень передбачуваності часу реагування на запит. Кожен запит повинен виконуватися впродовж певного періоду часу. Тобто алгоритм не повинен відкладати виконання запиту до безкінечності.

У наступних розділах розглядаються декілька широко застосованих алгоритмів диспетчеризації.

15.7.1 Алгоритм First Come First Served

Простим алгоритмом, до якого ми вже повинні були звикнути, є алгоритм **First Come First Served (FCFS)** – «першим прийшов, першим обслужений», або «першим увійшов – першим вийшов» (FIFO), що просто означає обробку запитів з черги в порядку їх надходження.

Алгоритм простий в реалізації, але може призводити до досить тривалого загального часу обслуговування запитів. Розглянемо приклад. Нехай у нас до диску з 40 циліндрів (від 0 до 39) є така черга запитів: 1, 36, 16, 34, 9, 12 і головки в початковий момент знаходяться на 11-му циліндрі. Тоді положення головок мінятиметься таким чином (рис. 15.8):

11→1→36→16→34→9→12

і всього головки перемістяться на **111** циліндрів (10+35+20+18+25+3).

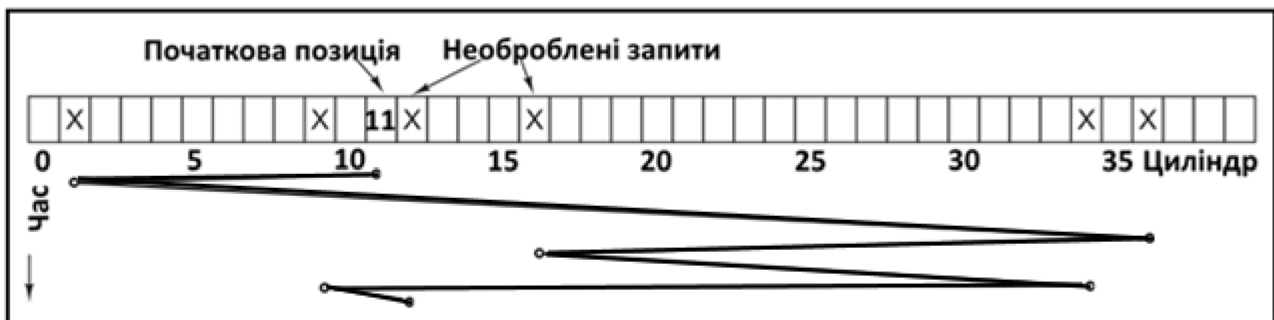


Рисунок 15.8 – Алгоритм планування FCFS

Неефективність алгоритму добре ілюструється двома останніми переміщеннями з 1 циліндра через увесь диск на 36 циліндр і потім знову з циліндра 9 на циліндр 34. При використанні FCFS сподіватися на високу продуктивність можна тільки при невеликій кількості процесів і запитах до близьких груп секторів. Із зростанням навантаження алгоритм FCFS швидко насичується, досягаючи граничної продуктивності, і час виконання запиту стає занадто великим (черга запитів швидко збільшується).

15.7.2 Алгоритм диспетчеризації SSTF

Алгоритм вибору найменшого часу обслуговування (Shortest Service Time First – **SSTF**) полягає у виборі того дискового запиту на введення-виведення, яке вимагає найменшого переміщення головок з поточної позиції. Отже, мінімізується час пошуку. Постійний вибір мінімального часу пошуку не дає гарантії, що середній час пошуку при усіх переміщеннях буде мінімальним, але, проте, ця стратегія забезпечує кращу в порівнянні з FIFO продуктивність. Оскільки головки можуть переміщатися в двох напрямах, то при рівних відстанях для ухвалення рішення може бути використаний випадковий вибір. Для попереднього прикладу (1, 36, 16, 34, 9, 12) цей алгоритм дасть таку послідовність положень головок (рис. 15.9):

11→12→9→16→1→34→36 (1+3+7+15+33+2)

і всього головки перемістяться на **61** циліндр (FCFS – 111).

Рисунок 15.9 – Алгоритм планування SSTF

Цей алгоритм зменшує загальні переміщення блоку головок в порівнянні з алгоритмом FCFS приблизно в два рази.

На жаль, алгоритм SSTF не обходиться без недоліків. Припустимо, що за час обробки запитів, показаних на рис. 15.9, продовжують надходити все нові і нові запити. Наприклад, якщо після переміщення до циліндра 16 є запит до циліндра 8, цей запит буде мати пріоритет над запитом до циліндра 1. Якщо потім надійде запит до циліндра 13, то в наступну чергу блок перейде до нього, а не до циліндра 1.

При високій завантаженості диска блок головок буде більшу частину часу залишатися в його середній частині, тому запитам до крайніх циліндрів доведеться чекати, поки статистичні відхилення в завантаженості диска не приведуть до відсутності запитів до середніх циліндрів. Запити, віддалені від середньої частини, можуть погано обслуговуватися. Тут цілі досягнення рівнодоступності і мінімізації часу відгуку вступають в конфлікт.

Алгоритм SSTF погано підходить для інтерактивних систем, які повинні надавати кожному користувачеві рівні, передбачувані часи реагування.

15.7.3 Алгоритм диспетчеризації SCAN

Усі стратегії, розглянуті раніше (окрім FIFO), можуть залишити деякий запит до тих пір, поки не звільнитися вся черга. Тобто при роботі завжди можуть бути нові запити, які будуть вибрані до вже наявного запиту в черзі. Уникнути такого роду «голодування» можна при використанні іншої стратегії.

У простому з алгоритмів сканування – **SCAN** – переміщення головки відбувається тільки в одному напрямі, задовольняючи ті запити, які відповідають вибраному напряму. Після досягнення останньої доріжки у вибраному напрямі (чи коли вичерпуються можливі запити), напрям змінюється на протилежний.

Нехай в попередньому прикладі в початковий момент часу головки рухаються у напрямі зменшення номерів циліндрів. Послідовність переміщення головок для запитів (12, 16, 34, 36, 39, 9, 1) виглядає таким чином:

11→12→16→34→36→39→9→1 (1+4+18+2+3+30+8)

і всього головки перемістяться на **66** циліндрів (FCFS – 111, SSTF – 61).

В данном случае алгоритм SCAN оказался немного хуже, чем SSF.

Але, якщо ми знаємо, що обслужили останній попутний запит у напрямі руху головок, то ми можемо не доходити до краю диска, а відразу змінити напрям руху на зворотній (рис. 15.10):

11→12→16→34→36→9→1 (1+4+18+2+27+8)

і всього головки перемістяться на **60** циліндри**.** Отримана модифікація алгоритму SCAN дістала назву **LOOK** (SCAN-LOOK).

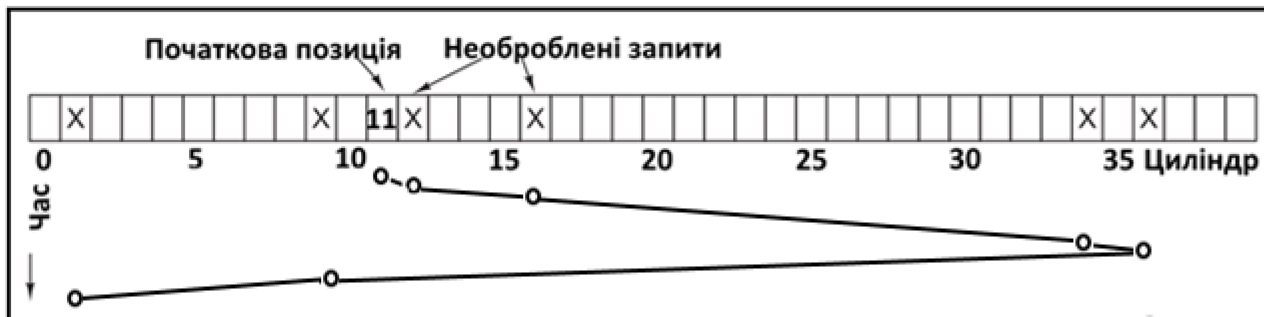


Рисунок 15.10 – Алгоритм планування SCAN-LOOK

Неважко побачити, що стратегія SCAN надає перевагу тим завданням, чиї запити стосуються доріжок, які знаходяться щонайближче до центру або найвіддалені від нього. Вона також може віддавати перевагу запитам, які поступили останніми.

15.7.4 Алгоритм диспетчеризації C-SCAN

Допустимо, що до моменту зміни безпосередньо руху головки в алгоритмі SCAN, тобто коли головка досягла одного з країв диска, у цього краю накопичилася велика кількість нових запитів, на обслуговування яких буде витрачено досить багато часу (не забуваємо, що треба не лише переміщати головку, але ще і передавати прочитані дані). Тоді запити, що належать до іншого краю диска і поступили раніше, чекатимуть обслуговування несправедливо довго. Для скорочення часу очікування запитів застосовується інша модифікація алгоритму SCAN – **C-SCAN** (circular SCAN – циклічне сканування) [12]. Коли головка досягає одного з країв диска, вона без читання попутних запитів (іноді істотно швидше, ніж при виконанні звичайного пошуку циліндра) переміщається на інший край, звідки знову починає рух у тому ж напрямі. Для цього алгоритму послідовність переміщень виглядатиме так (12, 16, 34, 36, 39, 9, 1):

11→12→16→34→36→39; 0→1→9 (1+4+18+2+3+1+8)

і всього головки перемістяться на **37** циліндрів.

По аналогії з алгоритмом LOOK для алгоритму SCAN також можна запропонувати метод **C-LOOK** для алгоритму C-SCAN (C-SCAN-LOOK):

11→12→16→34→36; 0→1→9 (1+4+18+2+1+8=**34**)

(FCFS – 111, SSTF – 61, SCAN – 66, SCAN – LOOK – 60)

Існують і зовсім інші алгоритми, але на цьому закінчимо наш огляд.

Контрольні питання і тести до розділу 15

Контрольні питання

1. На які два типи діляться пристрої введення-виведення?
2. З яких компонент складається зовнішній пристрій?
3. Які прості функції виконує контролер?
4. Які задачі повинна розв'язувати підсистема введення-виведення в мультипрограмній ОС при обміні даними із зовнішніми пристроями?
5. Що собою являє динамічне завантаження і вивантаження драйверів?
6. Як працюють операції введення-виведення в синхронному і асинхронному режимах?
7. Яка методика застосовується для зручного виконання читання даних заздалегідь, до реального запиту, а запис даних – трохи пізніше за реальний запит?
8. Опишіть схему роботи одинарного буфера для блочно-орієнтованих пристроїв.
9. Навіщо в ОС застосовується удосконалена схема одинарної буферизації шляхом використання двох і більше системних буферів?
10. Який механізм застосовується в ОС для звільнення процесора від операцій послідовного введення-виведення даних?

Тести

1. Для збільшення швидкості виконання додатків за необхідності пропонується використати … режим роботи введення-виведення.
    1) синхронний;
    2) пріоритетний;
    3) автоматичний;
    4) асинхронний.
2. Простим варіантом прискорення дискових операцій читання даних можна вважати використання подвійної:
    1) кластеризації;
    2) буферизації;
    3) диспетчеризації;
    4) пріоритизації.
3. Контролери введення-виведення …
    1) служать для підключення зовнішніх пристроїв введення-виведення до системної плати комп'ютера через відповідні порти;
    2) служать для відображення пристроїв введення-виведення в адресні простори системи;
    3) здійснюють передачу даних і команд процесора поза обчислювальною системою;
    4) управляють пристроями введення-виведення, прийомом і передачею даних через порти і виставлянням сигналів на магістралі.
4. Байт-орієнтовані пристрої це:
    1) клавіатура, миша, принтер, послідовний порт;

2) пристрої зберігання інформації, місткість яких може бути виміряна в байтах;
3) магнітний диск, оптичний диск;
4) пристрій, який підключається через порт USB.

5. Блок-орієнтований пристрій:
   1) клавіатура, миша, принтер, послідовний порт;
   2) пристрій, який підключається через порти введення/виведення;
   3) магнітний диск, оптичний диск;
   4) пристрій, який підключається через порт USB.
6. Контролер зовнішнього пристрою – це:
   1) система контролю прийому даних від зовнішнього пристрою;
   2) система управління зовнішнім пристроєм;
   3) система забезпечення цілісності даних, які зберігаються на зовнішньому пристрої зберігання інформації;
   4) електронний компонент для управління зовнішнім пристроєм і організації обміну даними.
7. Синхронна передача даних забезпечує:
   1) призупинення програми, яка дала запит на передачу;
   2) максимальну швидкість передачі даних;
   3) мінімальний час затримки при передачі даних;
   4) максимальну надійність передачі даних.
8. Що таке синхронне введення-виведення?
   1) введення-виведення, яке виконується завжди в один і той же час;
   2) введення-виведення, при виконанні якого програма призупиняється і чекає його закінчення;
   3) одночасне введення-виведення інформації з декількох паралельних процесів;
   4) введення-виведення, при виконанні якого програма продовжує роботу без очікування його закінчення.
9. Числове значення – 12, 16 або 32 – у файловій системі FAT означає:
   1) максимальний об'єм файлу в байтах;
   2) розмір кластера на диску;
   3) допустиму кількість символів в імені файлу;
   4) розрядність (розмір) елементу в таблиці FAT.
10. Згідно з яким із алгоритмів планування запитів до жорсткого диска всі запити обслуговуються в порядку їх надходження?
    1) SCAN;
    2) SSTF (Short Seek Time First);
    3) FCFS (First Come First Served);
    4) C-SCAN.
11. У якому з алгоритмів планування запитів до жорсткого диску в першу чергу обслуговуються запити, дані для яких лежать поряд з поточною позицією головки?
    1) SCAN;
    2) SSTF (Short Seek Time First);

3) FCFS (First Come First Served);
4) C-SCAN.

12. У якому з алгоритмів планування запитів до жорсткого диску головки постійно переміщаються від одного краю диска до іншого, по дорозі обслуговуючи всі запити, що зустрічаються, а після досягнення іншого краю напрям руху міняється і все повторюється знову?
    1) SCAN;
    2) SSTF (Short Seek Time First);
    3) FCFS (First Come First Served);
    4) C-SCAN.
13. У якому з алгоритмів планування запитів до жорсткого диску головки після досягнення одного з країв диска без читання попутних запитів переміщаються на інший край, звідки знову починають свій рух в колишньому напрямі?
    1) SCAN;
    2) SSTF (Short Seek Time First);
    3) FCFS (First Come First Served);
    4) C-SCAN.
14. На диску зі 100 циліндрів (від 0 до 99) є така черга запитів: 23, 67, 55, 14, 31, 7, 84, 10 і головки в початковий момент знаходяться на 63-циліндрі. Якому алгоритму планування запитів до жорсткого диска відповідає наступна послідовність положень головок:
    63→55→31→23→14→10→7→0→67→84?
    1) SCAN;
    2) SSTF (Short Seek Time First);
    3) FCFS (First Come First Served);
    4) C-SCAN.
15. На диску з 100 циліндрів (від 0 до 99) є наступна черга запитів: 23, 67, 55, 14, 31, 7, 84, 10 і головки в початковий момент знаходяться на 63-циліндре. Якому алгоритму планування запитів до жорсткого диска відповідає послідовність положень головок:
    63→67→55→31→23→14→10→7→84?
    1) SCAN;
    2) SSTF (Short Seek Time First);
    3) FCFS (First Come First Served);
    4) C-SCAN.
16. На диску з 100 циліндрів (від 0 до 99) є така черга запитів: 23, 67, 55, 14, 31, 7, 84, 10 і головки в початковий момент знаходяться на 63-цилиндре. Якому алгоритму планування запитів до жорсткого диска відповідає наступна послідовність положень головок:
    63→55→31→23→14→10→7→84→67?
    1) SCAN;
    2) SSTF (Short Seek Time First);
    3) FCFS (First Come First Served);
    4) C-SCAN.

17. Програма, що призначена для управління зовнішнім пристроєм і яка враховує усі його особливості, називається:
    1) контролером;
    2) супервізором;
    3) драйвером;
    4) DMA-контролером.
18. Абсолютна (фізична) адресація жорсткого диска для пошуку даних на диску вимагає завдання таких координат:
    1) фізичний номер сектора;
    2) порядковий номер сектора на жорсткому диску;
    3) ім'я логічного диска і відносний номер сектора;
    4) номер головки, номер циліндра, номер сектора.
19. Щільність запису на жорсткому диску росте тим більше:
    1) чим ближче доріжка до зовнішнього краю;
    2) вона завжди однакова;
    3) чим ближче доріжка до центру.
20. Найменша одиниця обміну даними дискового пристрою з оперативною пам'яттю, яка адресується, – це:
    1) кластер;
    2) доріжка;
    3) сектор;
    4) циліндр.
21. Нумерація доріжок на жорсткому диску починається з:
    1) 1 від центру до зовнішнього краю;
    2) 1 від зовнішнього краю до центру;
    3) 0 від центру до зовнішнього краю;
    4) 0 від зовнішнього краю до центру.
22. Нехай у нас є диск з 20 циліндрами (від 0 до 19). Час переміщення головки між сусідніми циліндрами складає 2 мс. У поточний момент часу головка знаходиться на 12-му циліндрі і рухається в бік збільшення номерів циліндрів. Скільки часу оброблятиметься послідовність запитів на читання циліндрів: 9, 13, 7, 1 алгоритмом First Come First Served (часом читання циліндрів і зміни безпосередньо руху головок нехтувати)?
    1) 48 мс;
    2) 42 мс;
    3) 38 мс;
    4) 36 мс.
23. Нехай у нас є диск з 20 циліндрами (від 0 до 19). Час переміщення головки між сусідніми циліндрами складає 2 мс. У поточний момент часу головка знаходиться на 12-му циліндрі і рухається в бік збільшення номерів циліндрів. Скільки часу оброблятиметься наступна послідовність запитів на читання циліндрів 9, 13, 7, 1 при стратегії вибору найменшого часу обслуговування (часом читання циліндрів і зміни безпосередньо руху головок нехтувати)?
    1) 30 мс;

2) 32 мс;
3) 34 мс;
4) 38 мс.

24. Нехай у нас є диск з 20 циліндрами (від 0 до 19). Час переміщення головки між сусідніми циліндрами складає 2 мс. У поточний момент часу головка знаходиться на 12-му циліндрі і рухається в бік збільшення номерів циліндрів. Скільки часу оброблятиметься послідовність запитів на читання циліндрів: 9, 13, 7, 1 алгоритмом сканування – **SCAN** (часом читання циліндрів і зміни безпосередньо руху головок нехтувати)?
    1) 40;
    2) 50;
    3) 54;
    4) 58.
25. Нехай у нас є диск з 20 циліндрами (від 0 до 19). Час переміщення головки між сусідніми циліндрами складає 2 мс. У поточний момент часу головка знаходиться на 12-му циліндрі і рухається в бік збільшення номерів циліндрів. Скільки часу оброблятиметься послідовність запитів на читання циліндрів: 9, 13, 7, 1 алгоритмом сканування – **C-SCAN** (часом читання циліндрів і зміни безпосередньо руху головок нехтувати)?
    1) 38;
    2) 36;
    3) 32;
    4) 30.
26. Нехай у нас є диск з 20 циліндрами (від 0 до 19). Час переміщення головки між сусідніми циліндрами складає 2 мс. У поточний момент часу головка знаходиться на 12-му циліндрі і рухається в бік збільшення номерів циліндрів. Скільки часу оброблятиметься послідовність запитів на читання циліндрів: 9, 13, 7, 1 алгоритмом сканування – **SCAN-LOOK** (часом читання циліндрів і зміни безпосередньо руху головок нехтувати)?
    1) 34;
    2) 32;
    3) 28;
    4) 26.
27. Нехай у нас є диск з 20 циліндрами (від 0 до 19). Час переміщення головки між сусідніми циліндрами складає 2 мс. У поточний момент часу головка знаходиться на 12-му циліндрі і рухається в бік збільшення номерів циліндрів. Скільки часу оброблятиметься послідовність запитів на читання циліндрів: 9, 13, 7, 1 алгоритмом сканування – **C-SCAN-LOOK** (часом читання циліндрів і зміни безпосередньо руху головок нехтувати)?
    1) 18;
    2) 20;
    3) 22;
    4) 24.

16 ФАЙЛОВА СИСТЕМА

Однією з найважливіших функцій операційної системи є організація роботи з даними. Вона реалізується засобами файлової системи (ФС). Для цього ОС підміняє фізичну структуру даних деякою зручною для користувача логічною моделлю. Логічна модель файлової системи матеріалізується у вигляді ієрархічної структури каталогів. Базовим елементом цієї моделі є файл, який як і файлова система в цілому, може характеризуватися як логічною, так і фізичною структурою.

Файлова система – це частина операційної системи, призначення якої полягає в тому, щоб забезпечити користувачеві зручний інтерфейс при роботі з даними, що зберігаються на диску, і забезпечити спільне використання файлів декількома користувачами і процесами.

Файлова система дозволяє програмам обходитися набором досить простих операцій для виконання дій над деяким абстрактним об'єктом, що представляє файл. При цьому програмістам не треба мати справи з деталями дійсного розташування даних на диску, буферизацією даних і іншими низькорівневими проблемами передачі даних. Усі ці функції файлова система бере на себе. Файлова система розподіляє дискову пам'ять, підтримує іменування файлів, відображає імена файлів у відповідних адресах зовнішньої пам'яті, забезпечує доступ до даних, підтримує захист і відновлення файлів.

16.1 Файли

Файл є набором однорідних записів. Файл розглядається користувачем і додатком як єдине ціле і звернення до нього здійснюється за його іменем. Файли можна створювати і видаляти. Користувачі і програми можуть мати право доступу до файлу як єдиного цілого. У деяких складніших системах управління доступ здійснюється на рівні запису, а іноді навіть і на рівні поля.

16.1.1 Імена файлів

Файли ідентифікуються іменами. Користувачі дають файлам символьні імена, при цьому враховуються обмеження ОС як на символи, так і на довжину імені. До недавнього часу ці межі були дуже вузькими. Так в популярній файловій системі FAT довжина імен обмежується відомою схемою 8.3 (8 символів – власне ім'я, 3 символи – розширення імені). Проте користувачеві набагато зручніше працювати з довгими іменами, оскільки вони дозволяють дати файлу дійсно мнемонічну назву, за якою навіть через досить великий проміжок часу можна буде згадати, що містить цей файл. Тому сучасні файлові системи, як правило, підтримують довгі символьні імена файлів.

При переході до довгих імен виникає проблема сумісності з раніше створеними додатками, що використовують короткі імена. Щоб додатки могли звертатися до файлів відповідно до прийнятих раніше угод, файлова система повинна уміти надавати еквівалентні короткі імена (псевдоніми) файлам, що мають довгі імена.

16.1.2 Типи файлів

Файли бувають різних типів: звичайні файли, спеціальні файли, файли-каталоги. ***Звичайні файли*** у свою чергу підрозділяються на текстові і двійкові.

Текстові файли складаються з рядків символів, представлених в ASCII-коді. Це можуть бути документи, тексти програм тощо. Текстові файли можна прочитати на екрані і роздрукувати на принтері. Двійкові файли не використовують ASCII-коди, вони часто мають складну внутрішню структуру, наприклад, об'єктний код програми або архівний файл.

Спеціальні файли – це файли, що асоціюються з пристроями введення-виведення, які дозволяють користувачеві виконувати операції введення-виведення, використовуючи звичайні команди запису у файл або читання з файлу. Спеціальні файли, так само як і пристрої введення-виведення, діляться на блок-орієнтовані і байт-орієнтовані.

Каталог – це, з одного боку, група файлів, об'єднаних користувачем виходячи з деяких міркувань (наприклад, файли, що містять програми), а з іншого боку – це файл, що містить системну інформацію про групу файлів, його складових.

У каталозі міститься список файлів, що входять в нього, і встановлюється відповідність між файлами і їх характеристиками (атрибутами). Каталоги можуть безпосередньо містити значення характеристик файлів, як це зроблено у файловій системі MS-DOS, або посилатися на таблиці, що містять ці характеристики, як це реалізовано в ОС UNIX. Каталоги можуть утворювати ієрархічну структуру за рахунок того, що каталог нижчого рівня може входити в каталог вищого рівня.

16.1.3 Атрибути файлів

Поняття «файл» включає не лише дані і ім'я, але і атрибути. Атрибути – це інформація, що описує властивості файлу. Приклади можливих атрибутів файлу:

- тип файлу (звичайний файл, каталог, спеціальний файл тощо);
- власник файлу і автор файлу;
- пароль для доступу до файлу;
- інформація про дозволені операції доступу до файлу;
- час створення, час останнього доступу і останньої зміни;
- поточний розмір файлу і максимальний розмір файлу;
- ознака «тільки для читання»;
- ознака «прихований файл»;
- ознака «системний файл»;
- ознака «архівний файл»;
- ознака «двійкова/символьна»;
- ознака «тимчасова» (видалити після завершення процесу);
- ознака блокування;
- довжина запису у файлі;
- покажчик на ключове поле в записі та довжина ключа.

Характеристики файлів можуть використовувати різні набори атрибутів. Наприклад, у файлових системах, що підтримують неструктуровані файли, немає необхідності використовувати два останні атрибути в наведеному списку, пов'язані зі структуризацією файлу. У розрахованій на одного користувача ОС в наборі атрибутів будуть відсутні характеристики, що мають стосунок до користувачів і захисту, такі як власник файлу, автор файлу, пароль для доступу до файлу, інформація про дозволений доступ до файлу.

Значення атрибутів файлів можуть безпосередньо міститися в каталогах, як це зроблено у файловій системі MS-DOS (рис. 16.1). На рисунку представлена структура запису в каталозі, що містить просте символьне ім'я і атрибути файлу. Наприклад, атрибути (ознаки файлу): R – тільки для читання; A – архівний потрібні для програм резервного копіювання, по ньому вони визначають потрібно копіювати файл або ні.

Ім'я файлу	Розширення	Атрибути					Резерв	Час	Дата	Номер першого кластера	Розмір
						Резерв					
8 байт	3 байт	1 байт					10 байт	2 байт	2 байт	2 байт	4 байт

Рисунок 16.1 – Атрибути файлів MS-DOS (32 байти)

Порожні 10 байт задіяні в Windows 98.

Поле час (16 розрядів) розбивається на три підполя: секунди – 5 біт (2^5=32 тому зберігаються з точністю до 2-х секунд); хвилини – 6 біт; години – 5 біт.

Поле дати (16 розрядів) розбивається на три підполя: день – 5 біт; місяць – 4 біта; рік – 7 біт (починається з 1980 р, тобто максимальний 2107 р.).

Теоретично розмір файлів може бути до 4 Гб (32 розряди).

Усі блоки файлу в записі не зберігаються, а зберігається тільки перший блок. Цей номер використовується як індекс для 64К (для FAT-16) елементів FAT-таблиці, що зберігається в оперативній пам'яті.

Залежно від кількості блоків на диску в системі MS-DOS застосовується три версії файлової системи FAT: FAT-12, FAT-16, FAT-32 – для адреси використовуються тільки 28 біт, тому правильніше назвати FAT-28.

Розмір блоку (***кластера***) має бути кратним 512 байт.

16.2 Логічна організація файлу

У загальному випадку дані, що містяться у файлі, мають деяку логічну структуру. Ця структура є базою при розробці програми, призначеної для обробки цих даних. Наприклад, щоб текст міг бути правильно виведений на екран, програма повинна мати можливість виділити окремі слова, рядки, абзаци тощо. Підтримка структури даних може бути або цілком покладена на додаток, або в тому або іншому ступені цю роботу може взяти на себе файлова система.

У першому випадку, коли всі дії, пов'язані зі структуризацією і інтерпретацією утримуваного файлу, цілком належать до ведення додатка, файл представляється у ФС неструктурованою ***послідовністю даних.*** Додаток формулює запити до файлової системи на введення-виведення, використовуючи загальні для всіх додатків системні засоби, наприклад, вказуючи зміщення від початку файлу і кількість байт, які необхідно зчитати або записати. Потік байт, що поступив до додатку, інтерпретується відповідно до закладеної в програмі логіки.

Модель файлу, відповідно до якої вміст файлу представляється неструктурованою послідовністю (потоком) байт, стала популярною разом з ОС UNIX, а тепер вона широко використовується в більшості сучасних ОС, у тому числі в MS-DOS, Windows NT/2000, NetWare. Неструктурована модель файлу дозволяє легко організувати розподіл файлу між декількома додатками: різні додатки можуть по-своєму структурувати і інтерпретувати дані.

Інша модель файлу, яка застосовувалася в ОС OS/360, DEC RSX і VMS, нині використовується досить рідко – це ***структурований файл***. У цьому випадку підтримка структури файлу доручається файловій системі. Файлова система бачить файл як упорядковану послідовність логічних записів. Додаток може звертатися до ФС із запитами на введення-виведення на рівні записів.

Логічний запис є найменшим елементом даних, яким може оперувати програміст при організації обміну із зовнішнім пристроєм. Навіть якщо фізичний обмін з пристроєм здійснюється великими одиницями, операційна система повинна забезпечувати програмістові доступ до окремого логічного запису.

Файлова система може використовувати два способи доступу до логічних записів: читати або записувати логічні записи послідовно (***послідовний доступ***) або позиціонувати файл на запис з указаним номером (***прямий доступ***).

Очевидно, що ОС не може підтримувати всі можливі способи структуризації даних у файлі, тому в тих ОС, в яких існує підтримка логічної структуризації файлів, вона існує для невеликого числа широко поширених схем логічної організації файлу (рис. 16.2).

До таких способів структуризації належить представлення даних у вигляді записів, довжина яких фіксована в межах файлу (рис. 16.2, *а).* У такому разі доступ до n-го запису здійснюється або шляхом послідовного читання (n-1) попередніх записів, або прямо за адресою, обчисленою за її порядковим номером. Наприклад, якщо L – довжина запису, то початкова адреса n-го запису рівна $L*n$. Відмітимо, що при такій логічній організації розмір запису фіксований у межах файлу, а записи в різних файлах, що належать одній і тій же файловій системі, можуть мати різний розмір.

Інший спосіб структуризації полягає в представленні даних у вигляді послідовності записів, розмір яких змінюється в межах одного файлу. Якщо розташувати значення довжин записів так, як це показано на рис. 16.2, *б*, то для пошуку потрібного запису система повинна послідовно прочитати усі попередні записи. Обчислити адресу потрібного запису за її номером при такій логічній організації файлу неможливо, а, отже, не може бути застосований ефективніший метод прямого доступу.

Рисунок 16.2 – Способи логічної організації файлів

Файли, доступ до записів яких здійснюється послідовно, за номерами позицій, називаються ***неіндексованими***, або ***послідовними***.

Іншим типом файлів є ***індексовані файли***, вони допускають швидший прямий доступ до окремого логічного запису. В індексованому файлі (рис. 16.2, *в)* записи мають одне або більше ключових (індексних) полів і можуть адресуватися шляхом вказівки значень цих полів.

Для швидкого пошуку даних в індексованому файлі передбачається спеціальна ***індексна таблиця***, в якій значенням ключових полів ставиться у відповідність адреса зовнішньої пам'яті. Ця адреса може вказувати або безпосередньо на шуканий запис, або на деяку область зовнішньої пам'яті, займану декількома записами, до числа яких входить шуканий запис. В останньому випадку говорять, що файл має ***індексно-послідовну*** організацію, оскільки пошук включає два етапи: прямий доступ за індексом до вказаної області диска, а потім послідовний перегляд записів у вказаній області (рис. 16.3). Ведення індексних таблиць бере на себе файлова система.

Для пошуку потрібного запису за його ключем спочатку виконується пошук в індексному файлі. Після того як в ньому знайдено найбільше значення ключа, яке не перевищує шукане, триває пошук в головному файлі. Наприклад, нехай послідовний файл (головний) містить 1 млн записів. Для пошуку певного ключового значення потрібні в середньому 0,5 млн операцій доступу до записів. Якщо створити індексний файл, що містить 1000 елементів, то знадобиться в середньому 500 операцій доступу до індексного файлу, після чого ще потрібні в середньому 500 операцій доступу до головного файлу.

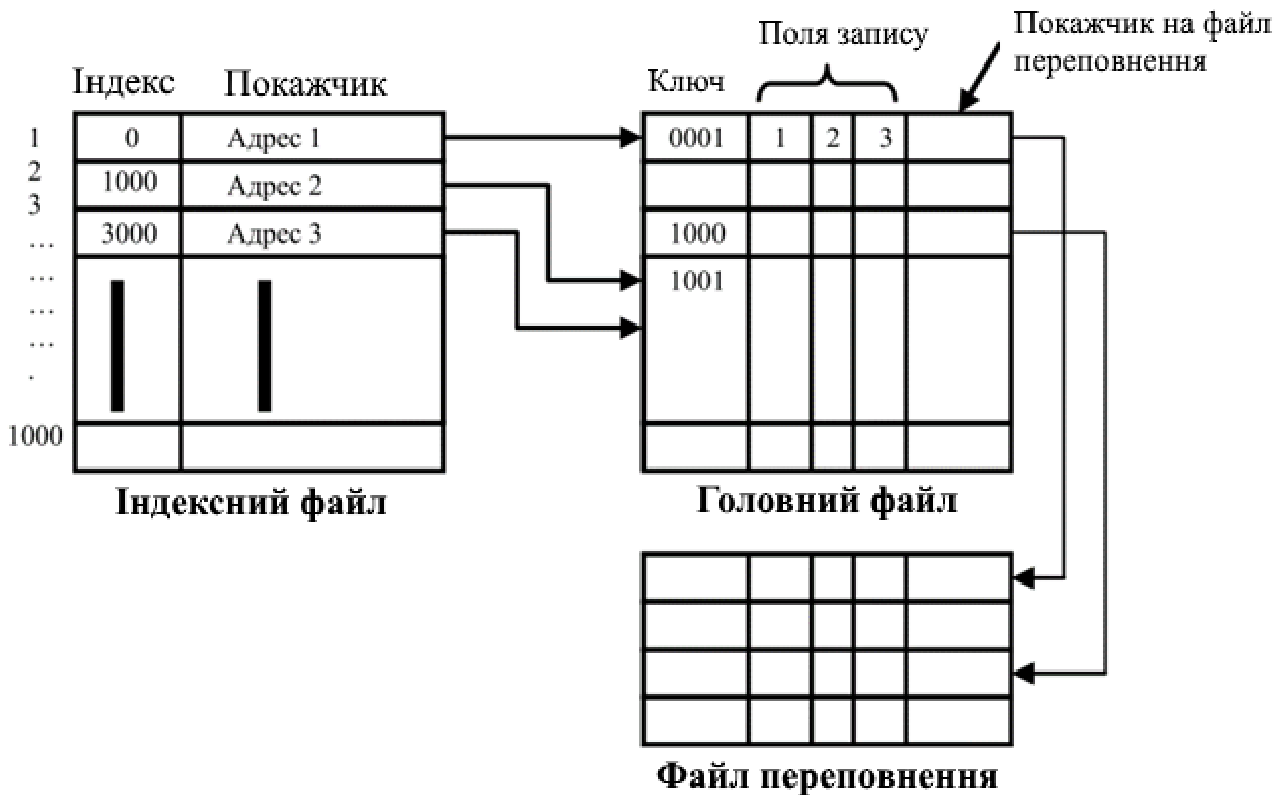


Рисунок 16.3 – Індексно-послідовний файл

У результаті середня довжина пошуку зменшується з 0,5 млн до 1 тис. Ще кращого результату можна досягти використовуючи багаторівневу індексацію. При цьому нижній рівень індексного файлу розглядається як послідовний файл, для якого створюється індексний файл верхнього рівня.

Доповнення до файлу обробляються таким чином. У кожному записі головного файлу міститься додаткове поле, яке невидиме для додатка і є покажчиком на файл переповнення. Якщо у файлі робиться вставка нового запису, вона додається у файл переповнювання. Запис в головному файлі, безпосередньо передуючий новому запису в логічній послідовності, оновлюється і вказує на новий запис у файлі переповнювання.

Час від часу виконується злиття індексно-послідовного файлу з файлом переповнювання.

16.3 Фізична організація файлу

Файл, що має образ цілісного набору байт, насправді дуже часто розкиданий «ділянками» по всьому диску, причому це розбиття ніяк не пов'язане з логічною структурою файлу. Наприклад, його окремий логічний запис може бути розташований в несуміжних секторах диска. Логічно об'єднані файли з одного каталогу зовсім не зобов'язані бути сусідами на диску. Принципи розміщення файлів, каталогів і системної інформації на реальному пристрої описуються фізичною організацією файлової системи.

16.3.1 Диски, розділи, сектори, кластери

Основним типом пристрою, який використовується в сучасних обчислювальних системах для зберігання файлів, є дискові накопичувачі. Ці пристрої призначені для зчитування і запису даних на жорсткі диски.

Ще раз нагадаємо топологію жорсткого диска, який складається з однієї або декількох скляних або металевих пластин, кожна з яких покрита з однієї або двох сторін магнітним матеріалом. Таким чином, диск у загальному випадку складається з пакету пластин (рис. 16.4).

На кожній стороні кожної пластини розмічені тонкі концентричні кільця – ***доріжки*** (traks), на яких зберігаються дані. Кількість доріжок залежить від типу диска. Нумерація доріжок починається з 0 від зовнішнього краю до центру диска. Коли диск обертається, елемент, що називається головкою, прочитує двійкові дані з магнітної доріжки або записує їх на магнітну доріжку.

Головка може позиціонуватися над заданою доріжкою. Головки переміщаються над поверхнею диска дискретними кроками, кожен крок відповідає зрушенню на одну доріжку. Запис на диск здійснюється завдяки здатності головки змінювати магнітні властивості доріжки. У деяких дисках уздовж кожної поверхні переміщається одна головка, а в інших – є по головці на кожну доріжку.

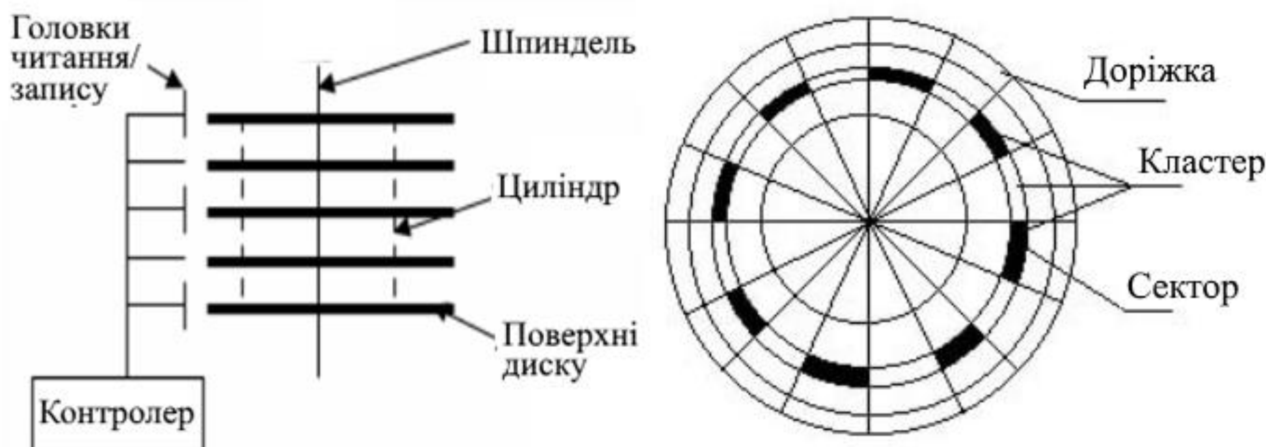


Рисунок 16.4 – Схема пристрою жорсткого диска

У першому випадку для пошуку інформації головка повинна переміщатися по радіусу диска. Зазвичай усі головки закріплені на єдиному переміщаючому механізмі і рухаються синхронно. Тому, коли головка фіксується на заданій доріжці однієї поверхні, усі інші головки зупиняються над доріжками з такими ж номерами. У тих же випадках, коли на кожній доріжці є окрема головка, ніякого переміщення головок з однієї доріжки на іншу не потрібно, за рахунок цього економиться час, що витрачається на пошук даних.

Сукупність доріжок одного радіусу на усіх поверхнях усіх пластин пакету називається ***циліндром*** (cylinder). Кожна доріжка розбивається на фрагменти, що називаються ***секторами*** (sectors), або ***блоками*** (blocks), так що усі доріжки мають рівне число секторів, в які можна максимально записати одне і те ж число байт. Іноді зовнішня доріжка має декілька додаткових секторів, які

використовуються для заміни пошкоджених секторів в режимі гарячого резервування.

Сектор має фіксований для конкретної системи розмір, що виражається степенем двійки. Найчастіше розмір сектора складає 512 байт. Враховуючи, що доріжки різного радіусу мають однакове число секторів, щільність запису стає тим вище, чим ближче доріжка до центру.

Для того щоб контролер міг знайти на диску потрібний сектор, необхідно задати йому усі складові адреси сектора: номер циліндра, номер поверхні і номер сектора. Оскільки прикладній програмі в загальному випадку потрібний не сектор, а деяка кількість байт, то типовий запит включає читання декількох секторів, що містять необхідну інформацію і надлишкові дані (рис. 16.5).

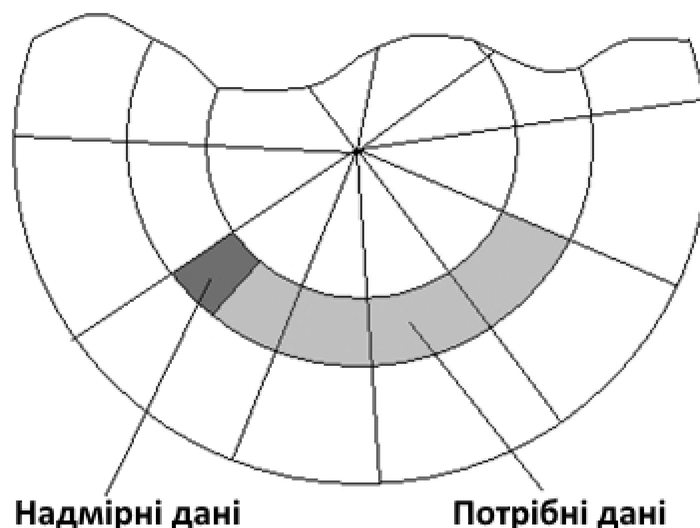


Рисунок 16.5 – Прочитування надмірних даних при обміні з диском

Операційна система при роботі з диском використовує, як правило, власну одиницю дискового простору, що називається ***кластером*** (cluster). Іноді кластер називають ***блоком*** (наприклад, в ОС Unix), що може призвести до термінологічної плутанини. Взагалі, термінологія, яка використовується при описі форматів дисків і файлових систем, залежить від апаратної платформи (RISC, Wintel тощо) і операційної системи. Це треба враховувати і трактувати терміни залежно від контексту.

При створенні файлу йому виділяється місце на диску кластерами. Наприклад, якщо файл має розмір 2560 байт, а розмір кластера у файловій системі визначений в 1024 байти, то файлу буде виділено на диску 3 кластери.

Розмітку диска під конкретний тип файлової системи виконують процедури високорівневого, або логічного форматування. При високорівневому форматуванні визначається розмір кластера і на диск записується інформація, що необхідна для роботи файлової системи, у тому числі інформація про доступний і невживаний простір, про межі областей, відведених під файли і каталоги, інформація про пошкоджені області. Крім того, на диск записується завантажувач ОС – невелика програма, яка починає процес ініціалізації операційної системи після включення живлення або рестарту комп'ютера.

Перш ніж форматувати диск під певну файлову систему, він може бути розбитий на розділи. ***Розділ*** – це неперервна частина фізичного диска, яку операційна система представляє користувачеві як логічний пристрій (використовуються також назви логічний диск і логічний розділ). У багатьох операційних системах використовується термін «***том***» (volume). У різних ОС тлумачення цього терміну має свої нюанси, але найчастіше він означає логічний пристрій, що відформатований під конкретну файлову систему.

Логічний пристрій функціонує так, якби це був окремий фізичний диск. Саме з логічними пристроями працює користувач, звертаючись до них за символьними іменами, використовуючи, наприклад, позначення A, B, C, SYS тощо.

На різних логічних пристроях одного і того ж фізичного диска можуть розташовуватися файлові системи різного типу. На рис. 16.6 показаний приклад диска, розбитого на три розділи, в яких встановлені дві файлові системи NTFS (розділи C і E) і одна файлова система FAT (розділ D).

Усі розділи одного диска мають однаковий розмір блоку, визначений для цього диска в результаті низькорівневого форматування. Проте в результаті високорівневого форматування в різних розділах одного і того ж диска, представлених різними логічними пристроями, можуть бути встановлені файлові системи, в яких визначені кластери різних розмірів.

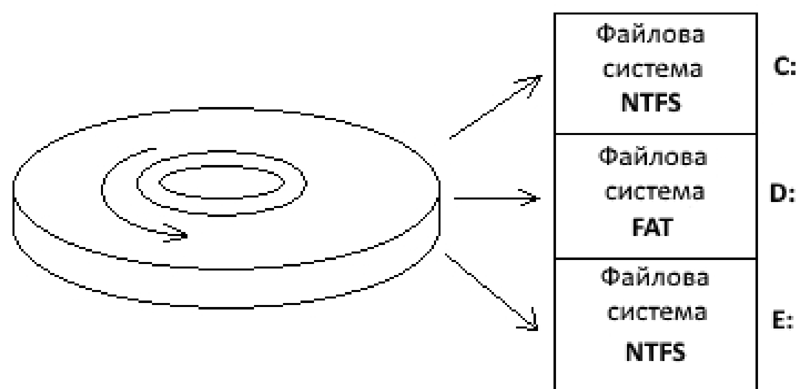


Рисунок 16.6 – Розбиття диска на розділи

16.3.2 Фізична організація і адресація файлу

Важливим компонентом фізичної організації файлової системи є фізична організація файлу, тобто спосіб розміщення файлу на диску. Основними критеріями ефективності фізичної організації файлів є:

- швидкість доступу до даних;
- об'єм адресної інформації файлу;
- міра фрагментації дискового простору;
- максимально можливий розмір файлу.

Безперервне розміщення – простий варіант фізичної організації (рис. 16.7, *а)*, при якому файлу надається послідовність кластерів диска, що утворюють безперервну ділянку дискової пам'яті. Основною перевагою цього методу є висока швидкість доступу, оскільки витрати на пошук і прочитування

кластерів файлу мінімальні. Також мінімальний об'єм адресної інформації – досить зберігати тільки номер першого кластера і об'єм файлу.

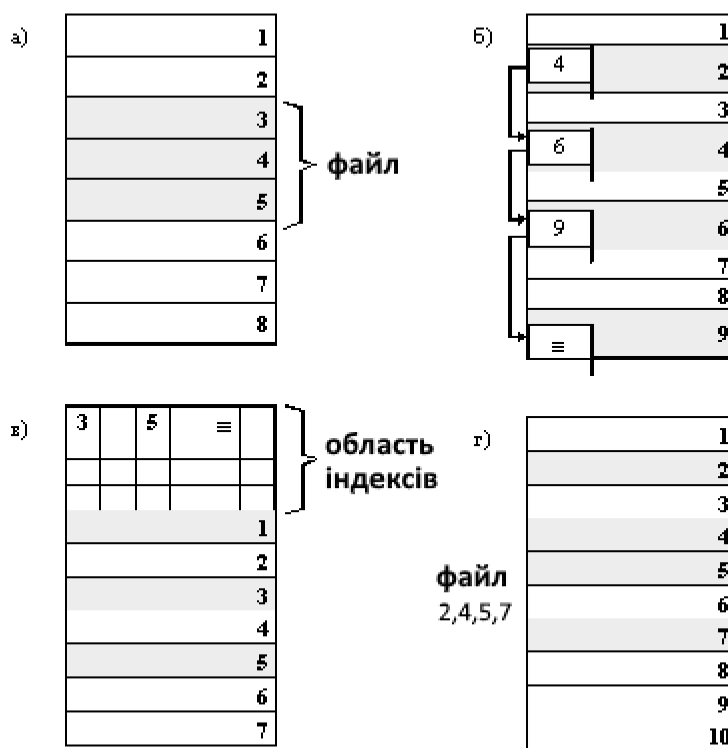


Рисунок 16.7 – Фізична організація файлу *а)* – безперервне розміщення; *б)* – зв'язаний список блоків; *в)* – зв'язаний список індексів; *г)* – перелік номерів блоків

Ця фізична організація не обмежує максимально можливий розмір файлу. Проте цей варіант має істотні недоліки, які утрудняють його застосування на практиці, незважаючи на всю його логічну простоту. Дійсно, якого розміру має бути безперервна область, що виділяється файлу, якщо файл при кожній модифікації може збільшити свій розмір? Ще серйознішою проблемою є фрагментація.

Через деякий час після створення файлової системи в результаті виконання численних операцій створення і видалення файлів простір диска неминуче перетворюється на «клаптеву ковдру», що включає велике число вільних областей невеликого розміру. При фрагментації, сумарний об'єм вільної пам'яті може бути дуже великим, а вибрати місце для розміщення цілого файлу неможливо.

Наступний спосіб фізичної організації – розміщення файлу у вигляді зв'язаного списку кластерів дискової пам'яті (див. рис. 16.7, *б*). При такому способі на початку кожного кластера міститься покажчик на наступний кластер. У цьому випадку адресна інформація мінімальна: розташування файлу може бути задане одним числом – номером першого кластера. На відміну від попереднього способу кожен кластер може бути приєднаний до ланцюжка кластерів якого-небудь файлу, отже, фрагментація на рівні кластерів відсутня.

Файл може змінювати свій розмір під час свого існування, нарощуючи число кластерів.

Недоліком є складність реалізації доступу до довільно заданого місця файлу. Щоб прочитати п'ятий за порядком кластер файлу, необхідно послідовно прочитати чотири перші кластери, простежуючи ланцюжок номерів кластерів. Крім того, при цьому способі кількість даних файлу, що містяться в одному кластері, не дорівнює степені двійки (одне слово витрачене на номер наступного кластера), а багато із програм читають дані кластерами, розмір яких дорівнює ступені двійки.

Популярним способом у файловій системі FAT, є використання зв'язного списку індексів (див. рис. 16.7, *в).* Цей спосіб є деякою модифікацією попереднього. Файлу виділяється пам'ять у вигляді зв'язного списку кластерів. Номер першого кластера запам'ятовується в записі каталогу, де зберігаються характеристики цього файлу. Інша адресна інформація відокремлена від кластерів файлу. З кожним кластером диска зв'язується деякий елемент – ***індекс***.

Індекси розташовуються в окремій області диска – в MS-DOS це таблиця FAT (File Allocation Table), що займає один кластер. Коли пам'ять вільна, всі індекси мають нульове значення. Якщо деякий кластер *N* призначений деякому файлу, то індекс цього кластера стає рівним або номеру *M* наступного кластера цього файлу, або набуває спеціального значення, що є ознакою того, що цей кластер для даного файлу є останнім. Індекс же попереднього кластера файлу набуває значення *N*, вказуючи на новопризначений кластер.

При такій фізичній організації зберігаються усі переваги попереднього способу: мінімальність адресної інформації, відсутність фрагментації, відсутність проблем при зміні розміру. Крім того, цей спосіб має додаткові переваги. По-перше, для доступу до довільного кластера файлу не вимагається послідовно прочитувати його кластери, досить прочитати тільки сектори диска, що містять таблицю індексів, відлічити потрібну кількість кластерів файлу по ланцюжку і визначити номер потрібного кластера. По-друге, дані файла заповнюють кластер цілком, тобто мають об'єм, рівний степені двійки.

Необхідно відмітити, що за відсутності фрагментації на рівні кластерів на диску все одно є певна кількість областей пам'яті невеликого розміру, які неможливо використати, тобто фрагментація все ж існує. Ці фрагменти є невживаними частинами останніх кластерів, призначених файлам, оскільки об'єм файлу в загальному випадку не кратний розміру кластера. На кожному файлі в середньому втрачається половина кластера. Це втрати особливо великі, коли на диску є велика кількість маленьких файлів, а кластер має великий розмір.

Ще один спосіб фізичного розташування файлу полягає в простому перерахуванні номерів кластерів, які зайняті цим файлом (див. рис. 16.7, *г*). Цей перелік і служить адресою файлу. Недолік цього способу очевидний: довжина адреси залежить від розміру файлу і для великого файлу може скласти значну величину. Перевагою ж є висока швидкість доступу до довільного кластера файлу, оскільки тут застосовується пряма адресація, яка виключає перегляд ланцюжка покажчиків при пошуку адреси довільного кластера файлу. Фрагментація на рівні кластерів у цьому способі також відсутня.

16.4 Файлова система FAT

Файлова система FAT (File Allocation Table) отримала своє найменування відповідно до назви методу організації даних – ***таблиці розподілу файлів***. Коли FAT була винайдена, це було чудове рішення для управління дисковим простором, головним чином тому, що гнучкі диски, на яких вона використовувалася, нечасто були розміром більше одного Мб. FAT була досить мала, і перебувала в пам'яті постійно, дозволяючи забезпечувати дуже швидкий довільний доступ до будь-якої частини будь-якого файлу.

Обмеження FAT на найменування файлів і каталогів успадковані з ОС CP/M. Нова видозмінена система, перейменована в MS-DOS (Microsoft Disk Operation System), майже повністю успадкувала структуру FAT від своєї попередниці.

FAT, спочатку орієнтована на невеликі диски і прості структури каталогу, використовувалася у всіх версіях MS-DOS і в перших двох випусках OS/2 (версії 1.0 і 1.1). Коли FAT була застосована на жорстких дисках, вона стала занадто великою для резидентного знаходження в пам'яті і погіршувала продуктивність системи. Крім того, використання FAT щодо великих кластерів на жорстких дисках призвело до великої кількості невикористаних ділянок, так як в середньому для кожного файлу половина кластеру була порожньою.

Протягом декількох років Microsoft і IBM робили спроби продовжити життя файлової системи FAT, завдяки зняттю обмежень на розміри тома, удосконалення стратегій розподілу, кешування імен шляху, і переміщення таблиць і буферів у розширену пам'ять.

Розмір таблиці FAT і розрядність використовуваних в ній індексних покажчиків визначається кількістю кластерів (блоків) в області даних. Для зменшення втрат через фрагментації бажано кластери робити невеликими, а для скорочення обсягу адресної інформації і підвищення швидкості обміну навпаки – чим більше кластер, тим краще. При форматуванні диска під файлову систему FAT зазвичай вибирається компромісне рішення, і розміри кластерів вибираються з діапазону від 1 до 128 секторів, або від 512 байт до 64 Кб.

У файлової системи FAT12 використовуються 12-розрядні покажчики, що дозволяє підтримувати до 4096 кластерів в області даних диска. Максимальний розмір розділу міг досягати 2 Мб – 2^{12} * 512 байт. Реально це число трохи менше, так як кілька значень індексного покажчика витрачається для ідентифікації спеціальних ситуацій, таких як «Останній кластер», «Невикористаний кластер», «Дефектний кластер» і «Резервний кластер». FAT-12 застосовувалася для гнучких дисків.

В FAT16 розміри кластерів – 512 байт, 1, 2, 4, 8, 16 і 32 Кб (2^{15}), і 16-розрядні покажчики для 65536 кластерів. Максимальний розмір розділу диска міг досягати 2 Гб (2^{16} * 32 Кб).

В FAT32 – 32-розрядні адреси для більш ніж 4 мільярдів кластерів (хоча в FAT32 використовується тільки 28 молодших бітів дискової адреси). Розміри кластерів 512 байт, 1, 2, 4, 8, 16 і 32 Кб. Максимальний розмір розділу диска міг

би досягати $2^{28} * 2^{15}$, але тут вже вступає інше обмеження – 512-байтні сектора адресуються 32-розрядних числом, а це $2^{32} * 2^{9}$, тобто 2 Тб.

Але все це в даний час можна розцінювати як тимчасові заходи, тому що файлова система просто вже не підходить до сучасних пристроїв довільного доступу.

На рис. 16.8 показана організація диска з використанням файлової системи FAT. FAT – це лінійна таблична структура з відомостями про файли: іменами файлів, їх атрибутами і іншими даними, що визначають місцезнаходження файлів в середовищі FAT. Елемент FAT визначає фактичну область диска, в якій зберігається початок фізичного файлу.

Блок параметрів BOOT-сектора (BR)	Резерв (ResSecs)	FAT1	FAT2 (копія)	Кореневий каталог (Rdir)	Каталоги і файли (область даних)

Рисунок 16.8 – Дисковий розділ FAT

У файлової системи FAT дисковий простір будь-якого логічного диска ділиться на дві області: системну область і область даних.

Системна область логічного диска створюється і ініціалізується при форматуванні, а згодом оновлюється при маніпулюванні файловою структурою. Область даних містить файли і каталоги, підпорядковані кореневому каталогу. Системна область складається з наступних компонентів, розташованих в логічному адресному просторі поспіль:

- стартовий сектор (сектор початкового завантаження, Boot-сектор);
- Boot Record складається з двох частин – disk parameter block (DPB) і system bootstrap (SB);
- структура блоку параметрів диска (DPB) служить для ідентифікації фізичного і логічного форматів логічного диска, а завантажувач system bootstrap (SB) грає істотну роль в процесі завантаження DOS.

Ця інформаційна структура (FAT-16) наведена в табл. 16.1:

Таблиця 16.1 – Структура завантажувального запису Boot Record для FAT-16

Позначення поля	**Довжина поля, байт**	**Вміст поля**
JAMP 3Eh	3, 00h	Безумовний перехід на початок SB
	8, 03h	Системний ідентифікатор
SectSize	2, 0Bh	Розмір сектора диска, байт
ClastSize	1, 0Dh	Кількість секторів в кластері
ResSecs	2, 0Eh	Кількість резервних секторів
FATcnt	1, 10h	Кількість копій FAT на диску
RootSize	2, 11h	Максим. к-ть елементів в каталозі RDir
TotSecs	2, 13h	Кількість секторів на диску
Media	1, 15h	Тип формату (Дескриптор) диска

Продовження таблиці 16.1

FATsize	2, 16h	Розмір FAT, секторів
TrkSecs	2, 18h	Кількість секторів на доріжці
HeadCnt	2, 1Ah	Кількість робочих поверхонь
HidnSecs	4, 1Ch	Кількість прихованих секторів
	4, 20h	Кількість секторів на диску, якщо розмір перевищує 32 Мб
	1, 24h	Тип диска (00h-гнучкий, 80h-жорсткий)
	1, 25h	Резерв
	1, 26h	Маркер з кодом 29h
	4, 27h	Серійний номер тому
	11, 2Bh	Мітка тому
	8, 36h	Ім'я файлової системи
	3Eh	System bootstrap (SB)
	2, 1FE	Сигнатура (слово AA55h)

Перші два байти boot record займає JMP-команда безумовного переходу в програму SB. Третій байт містить код 90h (NOP – немає операції). Далі розташовується 8-байтовий системний ідентифікатор, що включає інформацію про фірму розробника і версію ОС. Потім слідує DPB, а після нього SB.

16.4.1 Таблиця розміщення файлів

Таблиця розміщення файлів – дуже важлива інформаційна структура, що є картою (образом) області даних. Область даних розбивають на кластери. Кластер – це один або декілька секторів в логічному дисковому адресному просторі (точніше тільки в області даних). Кожному кластеру області даних відповідає один елемент FAT. У таблиці FAT кластери, що належать одному файлу (чи некореневому каталогу), зв'язуються в ланцюжки. Для вказання номера кластера в FAT-16 використовується 16-бітове слово (від 0 до 65535 кластерів). Файл або каталог займає ціле число кластерів, останній з яких може бути задіяний не в повному обсязі.

FAT – вкрай важливий елемент файлової структури**.** Порушення в FAT можуть призвести до повної або часткової втрати інформації на всьому логічному диску. Саме тому на диску зберігається дві копії FAT. Існують спеціальні програми, які контролюють стан FAT і виправляють порушення.

Кореневий Каталог – це певна область диска, що створюється в процесі ініціалізації (форматуванні) диска, де міститься інформація про файли і каталоги, що зберігаються на диску. Кореневий каталог завжди існує на відформатованому диску. На одному диску завжди тільки один кореневий каталог. Розмір кореневого каталогу для цього диска – величина фіксована, тому максимальна кількість «прив'язаних» до нього файлів і інших (дочірніх) каталогів (підкаталогів) – строго визначена.

Область даних диска представлена в MS DOS як послідовність пронумерованих кластерів.

Каталоги (підкаталоги) – спеціальні файли з 32-бітовими елементами для кожного файлу, що міститься в цьому каталозі. MS DOS підтримує ієрархічну структуру каталогів (деревоподібну).

Логічне розбиття області даних на кластери як сукупності секторів замість використання поодиноких секторів має такий сенс:

- зменшується розмір таблиці FAT;
- зменшується можлива фрагментація файлів;
- прискорюється доступ до файлу, оскільки в декілька разів скорочується довжина ланцюжків фрагментів дискового простору.

Проте занадто великий розмір кластера веде до неефективного використання області даних, особливо при великій кількості маленьких файлів. Елемент для кожного файлу включає:

- ім'я файлу (8+3 символи), байт атрибуту (8 біт);
- час модифікації (16 біт), дату модифікації (16 біт);
- перший розміщуваний блок (16 біт);
- розмір файлу (32 біта).

Ця інформація використовується всіма операційними системами, що підтримують файлову систему FAT. Біти атрибуту файлу в елементі каталогу вказують, чи має файл відповідні атрибути. Встановлений перший біт ідентифікує файл як підкаталог, а другий – в якості мітки тому. Призначеннями цих бітів управляє операційна система. Крім того, файли FAT мають чотири спеціальні атрибути, які вказують, як ці файли може застосовувати користувач: архівний, системний, прихований і тільки для читання.

Файлова система з використанням таблиці розміщення файлів FAT проілюстрована на рис. 16.9.

Початковий номер кластера записується в елемент каталогу (Directory Entry). Таблиця FAT має стільки елементів, скільки є кластерів на диску. В елемент таблиці FAT з номером, відповідним номеру кластера ланцюжка, записується номер наступного кластера ланцюжка. При форматуванні диска в елементи FAT, відповідні дефектним кластерам, записується код F7. Вільні кластери позначаються кодом 00. В елемент FAT, що відповідає останньому кластеру ланцюжка, записується код FF.

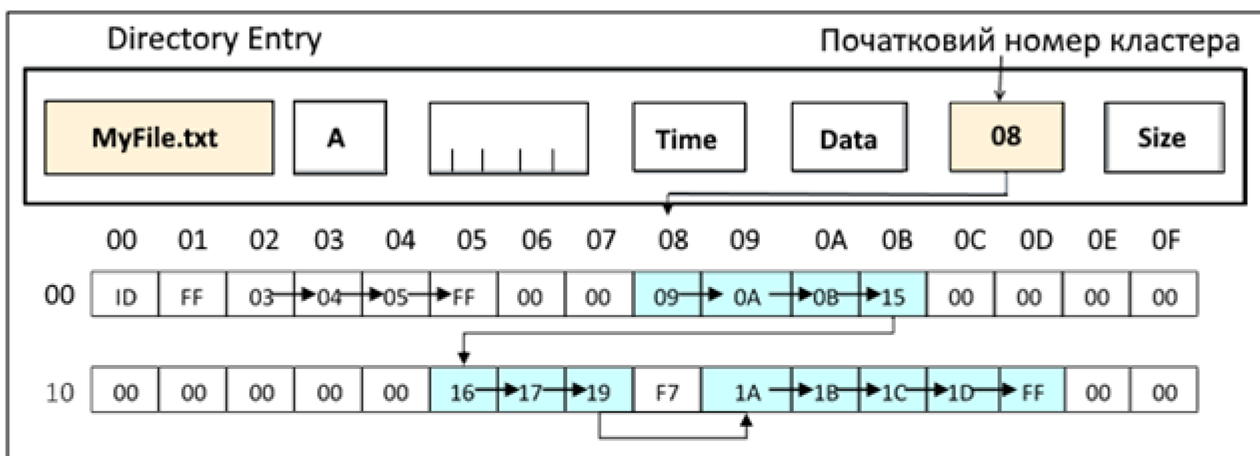


Рисунок 16.9 – Основна концепція FAT

З рис. 16.9 видно, що файл з ім'ям MyFile.txt розміщується, починаючи з 8-го кластера. Усього файл займає 12 кластерів. Ланцюжок кластерів (chain) для нашого прикладу: 8, 9, 0A, 0B, 15, 16, 17, 19, 1A, 1B, 1C, 1D. Кластер з номером 18 позначений спеціальним кодом F7 як поганий (bad), не використовується для розміщення даних. Кластер 1D позначений кодом FF як кінцевий кластер (останній в ланцюжку), що належить даному файлу. Вільні (незайняті) кластери позначаються кодом 00.

При виділенні нового кластера для запису файлу береться перший вільний кластер. Оскільки файли на диску змінюються – видаляються, переміщаються, збільшуються, зменшуються, – то згадане правило виділення першого вільного кластера для нової порції даних призводить до ***фрагментації файлів,*** що веде до істотного уповільнення роботи з файлами.

Windows NT, починаючи з версії 3.5, використовує біти атрибуту для підтримки довгих (до 255 символів) імен файлів в розділах FAT. Цей спосіб не заважає MS-DOS або OS/2 звертатися до подібного розділу. Всякий раз, коли користувач створює файл з довгим ім'ям (що перевищує стандартне для FAT обмеження «8+3»), Windows NT засновує елемент каталогу для цього файлу, що відповідає угоді «8+3» (за тими правилами, що і для NTFS) з додаванням одного або декількох вторинних елементів каталогу.

Кожен з таких вторинних елементів розрахований на 13 символів в довгому імені файлу і зберігає довгу частину імені файлу в UNICODE. Для цих елементів встановлюються атрибути: том, системний, прихований, тільки для читання. MS-DOS і OS/2 ігнорують елементи каталогу з таким набором атрибутів, і останні не видимі в цих операційних системах. Замість них MS-DOS і OS/2 звертаються до елементів, що містять інформацію в стандартному виді «8+3».

Файлова система FAT Windows 9x і Windows NT функціонує аналогічно MS-DOS. Windows NT можна встановлювати на існуючому розділі FAT. Якщо ж комп'ютер працює під управлінням Windows 9x, довгі імена файлів і каталогів цілком допустимі, оскільки механізми роботи з довгими іменами в обох системах однакові. Файлова система FAT не забезпечує захисту даних і їх автоматичного відновлення. Тому FAT використовується лише в тому випадку, якщо на комп'ютері в якості альтернативної системи встановлена MS-DOS або Windows 9x.

16.4.2 Файлова система FAT32

FAT32 – модифікована версія FAT, що дозволяє створювати розділи обсягом понад 2 Гб. Крім того, вона дає можливість використовувати кластери меншого розміру, і, відповідно, ефективніше витрачати дисковий простір. Вперше дана файлова система з'явилася в Windows 95 OSR2. У табл. 16.2 порівнюються розміри кластерів, що встановлюються за замовчуванням для FAT і FAT32.

Таблиця 16.2 – Співвідношення між розміром розділу і розміром кластерів

Об'єм диска	**Розмір кластера на FAT**	**Розмір кластера на FAT32**
0 Мб – 32 Мб	512 байт	
32 Мб – 64 Мб	1 Кб	
64 Мб – 127 Мб	2 Кб	
128 Мб – 255 Мб	4 Кб	
256 Мб – 511 Мб	8 Кб	
512 Мб – 1023 Мб	16 Кб	
1024 Мб – 2048 Мб	32 Кб	
260 Мб – 8 Гб		4 Кб
8 Гб – 16 Гб		8 Кб
16 Гб – 32 Гб		16 Кб
> 32 Гб		32 Кб

Формат нової файлової системи не сумісний з колишнім форматом FAT, тому слід уважно підходити до вибору для роботи з диском таких утиліт, як дефрагментатори, антивіруси тощо. Завантажувальні записи інших ОС відрізняються від розглянутої. Так, наприклад, в завантажувальному секторі для тома з FAT-32 у блоці DPB містяться додаткові поля, а ті поля, що знаходяться в звичному для системи FAT-16 місці, перенесені. Структура завантажувального запису для FAT-32 наведена в табл. 16.3.

Таблиця 16.3 – Структура завантажувального запису Boot Record для FAT-32

Позначення поля	**Довжина поля**	**Вміст поля**
JAMP 3Eh	3, 00h	Безумовний перехід на початок SB
	8, 03h	Системний ідентифікатор
SectSize	2, 0Bh	Розмір сектора диска, байт
ClastSize	1, 0Dh	Кількість секторів в кластері
ResSecs	2, 0Eh	Кількість резервних секторів, для FAT-32 – 32
FATcnt	1, 10h	Кількість копій FAT на диску
RootSize	2, 11h	Максим. кількість елементів в каталозі Rdir (0000h)
TotSecs	2, 13h	Кількість секторів на диску (0000h)
Media	1, 15h	Тип формату (Дескриптор) диска
FATsize	2, 16h	Размір FAT, секторів (0000h)
TrkSecs	2, 18h	Кількість секторів на доріжці
HeadCnt	2, 1Ah	Кількість робочих поверхонь
HidnSecs	4, 1Ch	Кількість прихованих секторів(розташовуються перед завантажувальним сектором). Використовуються при завантаженні обчислення абсолютного зміщення кореневого каталогу і даних

Продовження таблиці 16.3

	4, 20h	Кількість секторів на диску
	4, 24h	Число секторів в таблиці FAT
	2, 28h	Розширені прапори
	2, 2Ah	Версія файлової системи
	4, 2Ch	№ кластера для 1-го кластера кореневого каталогу
	2, 34h	№ сектора з резервною копією завантажувального сектора
	12, 36h	Зарезервовано

16.5 Файлова система NTFS

В порівнянні з FAT або FAT32, NTFS (New Technology File System) надає користувачеві ціле поєднання переваг: ефективність, надійність і сумісність. Перечислимо деякі її особливості.

1. 64-розрядні адреси, тобто теоретично може підтримувати $2^{64}*2^{16}$ байт (18 446 744 073 Тб).
2. Розміри блоку (кластеру) від 512 байт до 64 Кб, частіше використовується 4 Кб.
3. Підтримка великих файлів.
4. Імена файлів обмежені 255 символами Unicode.
5. Довжина шляху обмежується 32767 (2^{15}) символами Unicode.
6. Імена чутливі до регістра, my.txt і MY.TXT це різні файли.
7. Файлова система з журналом, тобто не потрапить в суперечливий стан після збоїв.
8. Контроль доступу до файлів і каталогів.
9. Підтримка жорстких і символічних посилань.
10. Підтримка стискування і шифрування файлів.
11. Підтримка дискових квот.

Система NTFS розроблена для швидкого виконання операцій як стандартних файлових (читання, запис і пошук), так і удосконалених (наприклад, відновлення файлової системи) на дуже великих жорстких дисках.

Підтримуючи управління доступом до даних, NTFS дає гарантії безпеки, необхідні для файлових серверів і персональних комп'ютерів в корпоративному середовищі. Це важливо для цілісності корпоративних даних.

Система NTFS проста, але дуже потужна розробка, для якої вся інформація на томі NTFS – файл або частина файлу. Кожен розподілений на томі NTFS сектор належить деякому файлу. Частиною файлу є навіть метадані файлової системи (інформація, що описує безпосередньо файлову систему). Файлова система підтримує об'єктно-орієнтовані додатки, обробляючи всі файли як об'єкти з атрибутами, обумовленими користувачем і системою.

16.5.1 Ключові можливості NTFS

До основних особливостей NTFS належать:

1. **Здатність відновлення даних**. Це досягається за допомогою використання моделі обробки транзакцій для операцій обміну у файловій системі. Кожен обмін розглядається як атомарна дія, яка або виконується повністю, або не виконується зовсім. Окрім цього, NTFS використовує надмірне зберігання критичних даних файлової системи.
2. **Безпека**. Для забезпечення безпеки NTFS використовує об'єктну модель W2K. Відкритий файл реалізується як файловий об'єкт з дескриптором, що визначає атрибути безпеки.
3. **Диски і файли великих об'ємів**. NTFS підтримує дуже великі диски і файли ефективніше, ніж FAT і інші системи.
4. **Узагальнена індексація**. NTFS зв'язує з кожним файлом набір атрибутів. Набір описів файлів в системі управління файлами організований як реляційна база даних, тому файли можуть бути індексовані за будь-яким атрибутом.
5. **Множинні потоки даних**. Файл в NTFS – це не просто лінійна послідовність байтів, як файли в системах FAT-32 і UNIX. Замість цього файл складається з декількох атрибутів, кожен з яких представлений у вигляді потоку байтів. Більшість файлів мають декілька коротких потоків, таких як ім'я файлу і його 64-бітовий ідентифікатор, плюс один довгий (неіменований) потік з даними. У файлу може бути і декілька довгих потоків даних.

16.5.2 Том NTFS і файлова структура

NTFS використовує такі концепції дискового зберігання:

1. ***Сектор***. Найменша одиниця фізичного зберігання на диску. Розмір даних у байтах є степенем двійки і майже завжди рівний 512 байт.
2. ***Кластер***. Один або декілька послідовних секторів на одній доріжці. Звернення до блоків (кластерів) здійснюється за їх зміщенням від початку тому.
3. ***Том.*** Логічний розділ диска, що складається з деякої кількості кластерів. Том може займати як увесь диск, так і його частину або охоплювати декілька дисків (до 2^{64} байт).

Кластер є фундаментальною одиницею розміщення у файловій системі NTFS, яка не розпізнає сектори. Нині максимальний розмір файлу складає 2^{48} байт. Використання кластерів при розміщенні файлів робить систему NTFS незалежною від розмірів фізичних секторів. Це дозволяє їй без перешкод підтримувати нестандартні диски з розміром сектора, не рівним 512 байт. У таблиці. 16.4 приведені розміри кластерів системи NTFS за умовчанням.

Таблиця 16.4 – Розділи NTFS і розміри кластерів

Розмір тома	**Секторів у кластері**	**Розмір кластера**
<= 512 Мб	1	512 байт
512 Мб – 1 Гб	2	1 Кб
1 Гб – 2 Гб	4	2 Кб
2 Гб – 4 Гб	8	4 Кб
4 Гб – 8 Гб	16	8 Кб
8 Гб – 16 Гб	32	16 Кб
16 Гб – 32 Гб	64	32 Кб
> 32 Гб	128	64 Кб

16.5.3 Схема тома NTFS

NTFS використовує простий і в той же час потужний підхід в організації інформації на томі диска. Кожен елемент тома є файлом, і кожен файл складається з набору атрибутів (навіть дані, що зберігаються у файлі, розглядаються як атрибут). При такій простій структурі достатньо невеликої кількості функцій загального призначення для організації і управління файловою системою. На рис. 16.10 показана схема тому, що складається з 4-х областей.

Завантажувальний сектор розділу	Головна файлова таблиця	Системні файли	Область файлів

Рисунок 16.10 – Схема тома NTFS

Перші декілька секторів будь-якого тому займає **завантажувальний сектор розділу (**розміром до 16 секторів), що містять інформацію про схему тому і структури файлової системи, а також початкову завантажувальну інформацію і код завантаження. Потім іде ***головна файлова таблиця MFT*** (Master File Table), що є лінійною послідовністю записів фіксованого розміру (1 Кб). Вона містить інформацію про всі файли і каталоги цього тому, а також інформацію про вільний простір. За областю MFT іде область, завдовжки 1 Мб, що містить ***системні файли*** (див. рис. 16.10).

16.5.4 Головна файлова таблиця

Кожен файл на томі NTFS представлений записом фіксованого розміру в спеціальному файлі – ***головній файловій таблиці MFT (Master File Table)***. NTFS резервує перші 16 записів таблиці для спеціальної інформації, файлів метаданих (рис. 16.11).

Перший запис таблиці описує безпосередньо головну файлову таблицю. За нею йде дзеркальний запис MFT. Якщо перший запис MFT зруйнований, NTFS прочитує другий запис, щоб відшукати дзеркальний файл MFT, перший запис якого ідентичний першому запису MFT.

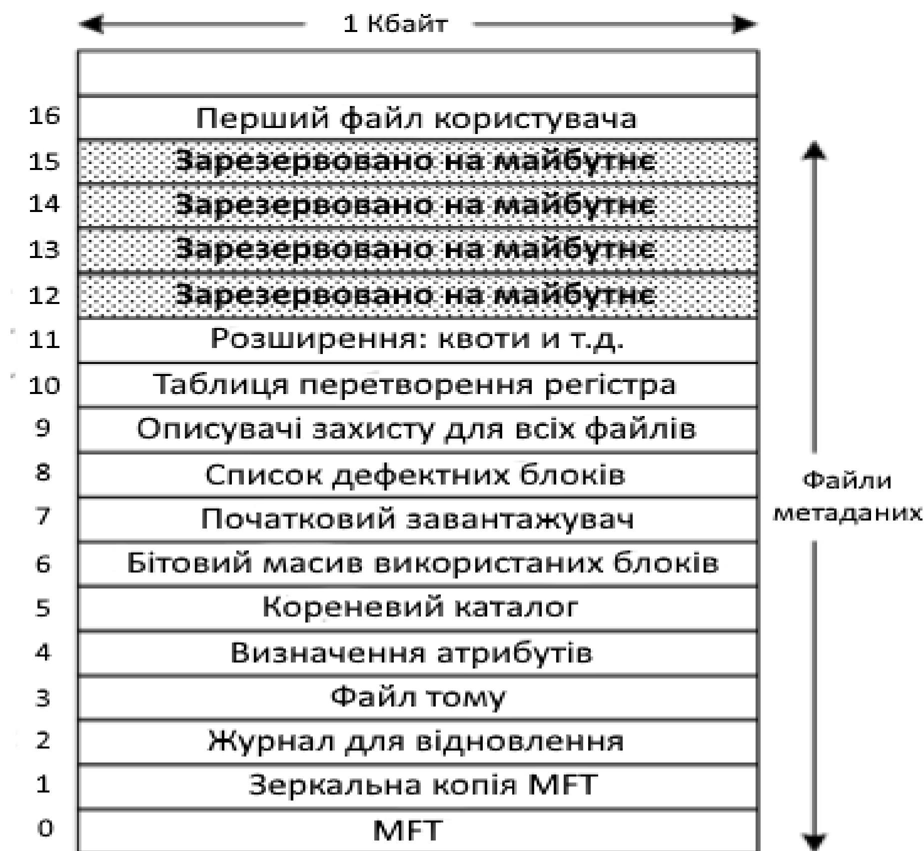


Рисунок 16.11 – Головна файлова таблиця MFT, кожен запис посилається на файл або каталог

Місце розташування сегментів даних MFT і дзеркального файлу MFT записане в секторі початкового завантаження. Дублікат сектора початкового завантаження знаходиться в логічному центрі диска. Третій запис MFT – файл реєстрації, який застосовується для відновлення файлів.

Сімнадцятий і подальші записи головної файлової таблиці використовуються файлами і каталогами на томі. Призначення цих файлів описане в показаній нижче таблиці MFT (табл. 16.5), що забезпечує дуже швидкий доступ до файлів.

Якщо файл дуже великий, то іноді використовується два і більше записів головної файлової таблиці, щоб вмістити список усіх блоків файлу. В цьому випадку 1-й запис MFT, що називається ***базовим записом***, вказує на інші записи MFT.

Сама головна файлова таблиця є файлом і, як будь-який файл, може розташовуватися в будь-якому місці тому, тим самим усувається проблема дефектних секторів на першій доріжці дискового розділу.

Кожен атрибут розпочинається із заголовка, що ідентифікує цей атрибут і значення, що повідомляє довжину, оскільки деякі атрибути, наприклад, ім'я файлу або дані, можуть мати змінну довжину. Якщо значення атрибуту занадто довге, воно розташовується в іншому місці диска, а в запис MFT поміщається покажчик на нього.

Таблиця 16.5 – Головна файлова таблиця NTFS (0-15 – файли метаданих)

Номер запису	**Системний файл**	**Ім'я файлу**	**Призначення файлу**
0	Головна таблиця	$Mft	Містить повний список файлів тому NTFS
1	Копія головної таблиці файлів	$MftMirr	Дзеркальна копія перших трьох записів MFT
2	Файл журналу	$LogFile	Список транзакцій, який використовується для відновлення файлової системи після збоїв
3	Том	$Volume	Ім'я тому, версія NTFS, інформація про том
4	Таблиця атрибутів	$AttrDef	Таблиця імен, номерів і описів атрибутів
5	Індекс кореневого каталогу	$.	Кореневий каталог
6	Бітова карта кластерів	$Bitmap	Розмітка використаних кластерів тому
7	Завантажувальний сектор розділу	$Boot	Адреса завантажувального сектора розділу
8	Файл поганих кластерів	$BadClus	Файл, що містить список усіх виявлених на томі поганих кластерів
9	Таблиця квот	$Quota	Квоти використовуваного простору на диску для кожного користувача
10	Таблиця перетворення регістра символів	SUpcase	Використовується для перетворення регістра символів для кодування Unicode
11	Розширення		Квоти
12-15	Зарезервовані		
16 …			Перший файл користувача і так далі

Перші 16 записів MFT зарезервовані для файлів метаданих NTFS. Кожен запис описує нормальний файл, в якого є атрибути і блоки даних, як у будь-якого файлу. У кожного такого файлу є ім'я, яке розпочинається з символу долара, що вказує на те, що це файл метаданих.

Перший запис описує сам файл MFT. Зокрема, він містить інформацію про розташування блоків файлу MFT, що дозволяє системі знайти файл MFT. Очевидно, щоб знайти всю іншу інформацію про систему, в ОС має бути деякий спосіб знаходження першого блоку файлу MFT, який міститься в завантажувальному блоці, куди він поміщається при розмітці тому.

Запис 1 є дублікатом першої частини файлу MFT. Ця інформація є настільки цінною, що наявність другої копії може бути потрібна, якщо один з перших блоків головної файлової таблиці раптом стане дефектним.

Запис 2 є журналом. Коли у файловій системі робляться структурні зміни, такі як додавання нового каталогу або видалення каталогу, інформація про майбутню операцію реєструється в журналі. Таким чином, збільшується ймовірність коректного відновлення файлової системи в разі збою під час виконання операції.

Запис 3 містить інформацію про том, наприклад, його розмір, мітка.

Запис 4 містить файл $AttrDef, в якому визначаються атрибути.

Запис **5** містить дані про кореневий каталог, який сам є файлом і може довільно збільшуватися в розмірах.

Запис 6 враховує вільне місце на диску за допомогою бітового масиву, що є також файлом.

Запис 7 вказує на файл початкового завантаження.

Запис 8 використовується для того щоб зв'язати разом усі дефектні блоки і гарантувати, що вони ніколи не зустрінуться у файлах.

Запис 9 містить інформацію про захист.

Запис 10 використовується для перетворення регістра. Для символів латинського алфавіту від A до Z перетворення регістра не представляє проблем. Для інших мов (наприклад, грузинської, грецької) це питання не таке очевидне, тому файл містить необхідні інструкції.

Запис 11 є каталогом, що містить різні файли для дискових квот, ідентифікаторів об'єктів тощо.

Кожен запис MFT має ***заголоовк запису***, за яким іде послідовність пар (заголовок атрибуту, значення). Заголовок запису містить:

- магічне число, яке використовується для перевірки дійсності запису;
- порядковий номер, що оновлюється кожного разу, коли запис використовується для нового файлу;
- лічильник звернень до файлу;
- кількість байт, використовуваних в записі;
- ідентифікатор (індекс, порядковий номер) базового запису, який використовується тільки для запису розширення, а також інші поля.

У файловій системі NTFS визначені 13 атрибутів, які можуть з'являтися в записах MFT (табл. 16.6).

Всі записи таблиці MFT складаються з послідовності заголовків атрибутів, кожен з яких ідентифікує наступний за ним атрибут, а також містить довжину і розташування поля значення з різноманітними прапорами та іншою інформацією.

Як правило, значення атрибутів розташовуються відразу ж за заголовками. Але, якщо довжина значення занадто велика, щоб поміститися в запис таблиці MFT, вона може бути поміщена в окремий блок диска. Такий атрибут називають ***нерезидентним атрибутом***. Наприклад, таким атрибутом є атрибут даних.

Ім'я потоку даних, якщо він є присутнім, міститься в заголовку атрибуту даних. Слідом за цим заголовком розташовується або список дискових адрес, що визначає місце розташування файлу на диску, або, для файлів завдовжки всього в декілька сотень байтів, сам файл.

Таблиця 16.6 – Типи атрибутів файлів і каталогів в NTFS

Атрибут	Опис
Стандартна інформація	Біти прапорів, тимчасові штампи тощо
Ім'я файлу	У кодуванні Unicode
Описувач захисту	Застарів. Тепер знаходиться в атрибуті $Extend$Secure
Список атрибутів	Розташування додаткових записів MFT
Ідентифікатор об'єкту	64-розрядний ідентифікатор файлу
Точка повторного аналізу	Для монтування і символьних посилань
Назва тому	Назва тому (тільки у $Volume)
Інформація про том	Версія тому (тільки у $Volume)
Кореневий індекс	Використовується для каталогів
Розміщення індексу	Використовується для дуже великих каталогів
Потік даних утиліти реєстрації	Управляє реєстрацією у файлі
Дані	Потокові дані; можуть повторюватися

Ім'я потоку даних, якщо він є присутнім, міститься в заголовку атрибуту даних. Слідом за цим заголовком розташовується або список дискових адрес, що визначає місце розташування файлу на диску, або, для файлів завдовжки всього в декілька сотень байтів, сам файл. Метод поміщення самого утримуваного файлу в запис MFT називається ***безпосереднім файлом.*** Як правило, всі дані файлу не поміщаються в запис MFT. Тому дискові блоки файлам призначаються заможливістю у вигляді ***серій послідовних блокі в (сегментів файлів)***. В ідеалі файл має бути записаний в одну серію (нефрагментований файл), файл, що складається з *n* блоків, може бути записаний від 1 до *n* серій.

На рисунку 16.12 представлений запис MFT для 9-блокового файлу, що складається з трьох сегментів (серій).

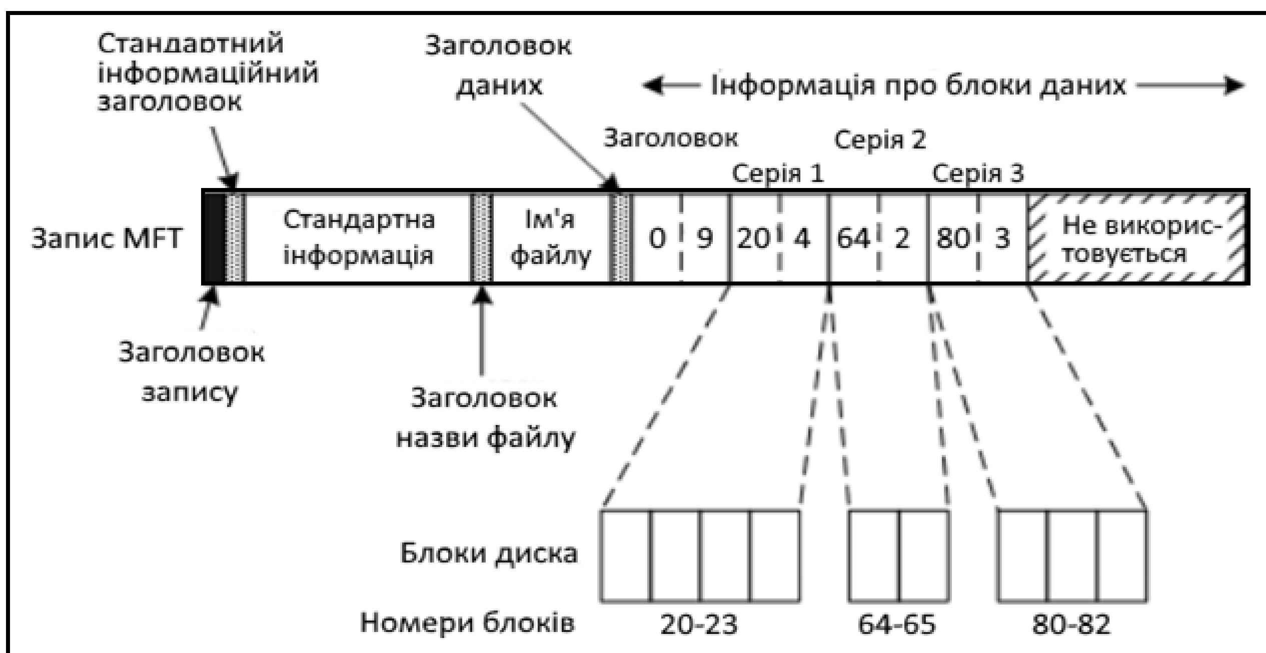


Рисунок 16.12 – Запис MFT для 9-блочного файлу

Увесь запис поміщається в один запис MFT. Заголовок містить кількість блоків (9 блоків). Кожна серія записується у вигляді пари, дискова адреса – кількість блоків (20-4, 64-2, 80-3). Кожна пара, за відсутності стискування, це два 64-розрядні числа (16 байт на пару). Багато адрес містять велику кількість нулів, стискування робиться за рахунок прибирання нулів в старших байтах. У результаті для пари потрібно частіше всього 4 байти.

16.5.5 Цілісність даних і відновлення в NTFS

NTFS – це відновлювана файлова система. Кожна операція введення-виведення, що змінює файл на томі NTFS, розглядається файловою системою як транзакція і може виконуватися як неподільний блок. При модифікації файлу користувачем сервіс файлу реєстрації фіксує всю інформацію, необхідну для повторення або відкату транзакції. Якщо транзакція завершена успішно, робиться модифікація файлу. Якщо ні, NTFS виконує відкат транзакції згідно інструкціями в інформації скасування. Якщо в транзакції виявлена помилка, то вона виконується в зворотному порядку.

При збої системи NTFS виконує три проходи: аналізу, повторів і відкатів. У процесі аналізу на підставі інформації файлу реєстрації NTFS оцінює ушкодження і точно визначає, які кластери треба модифікувати. Під час повторного проходу виконуються всі етапи транзакції від останньої контрольної точки. При відкаті відбувається повернення всіх незакінчених транзакцій.

Важлива особливість NTFS – ***відкладена передача,*** що дозволяє, подібно до відкладеного запису, мінімізувати витрати на реєстрацію транзакцій. Замість використання ресурсів для негайної відмітки транзакції як успішно завершеної, ця інформація заноситься в кеш і записується у файл реєстрації як фоновий процес. Якщо збій відбувається до того, як інформація про транзакцію була зареєстрована, NTFS зробить повторну перевірку транзакції для визначення її успішності. Якщо NTFS не може гарантувати, що транзакція завершилася успішно, робиться відкат транзакції. Ніякі незавершені модифікації тома не дозволені.

Кожні декілька секунд NTFS перевіряє кеш, щоб визначити стан відкладеного запису і відмітити його у файлі реєстрації як контрольну точку. Якщо вслід за визначенням контрольної точки наступить збій, система має можливість повернутися до стану, зафіксованого контрольною точкою. Цей метод призначений для захисту метаданих і забезпечує оптимальний час відновлення, зберігаючи чергу подій, яка може знадобитися в процесі відновлення. Дані користувача в разі збою системи можуть бути зруйновані.

16.5.6 Компресія файлів і каталогів

Особливість NTFS – можливість динамічного стискування файлів і каталогів. Файлова система NTFS підтримує прозоре стискування файлів. Файл може бути створений в стислому режимі. Це означає, що система NTFS автоматично намагатиметься стиснути блоки цього файлу при записі їх на диск і автоматично розпаковувати їх при читанні.

Стискування є новим атрибутом файлу або каталогу і подібно до будь-якого атрибуту може бути зняте або встановлене в будь-який момент часу. Стискування даних файлу відбувається таким чином. Коли система NTFS записує на диск файл, помічений для стискування, вона вивчає перших 16 логічних блоків файлу. Потім до цих блоків застосовується алгоритм стискування. Якщо отримані на виході блоки можуть поміститися в 15 або менше блоків, то стиснуті дані записуються на диск. Якщо отримати виграш хоч би в один блок не вдається, то дані 16 блоків так і записуються в нестиснутому виді. Потім увесь алгоритм повторюється для наступних 16 блоків і так далі.

16.6 Файлова система ReFS

Спочатку в Windows Server, а тепер і в Windows 10 з'явилася нова сучасна файлова система **ReFS** (Resilient File System – стійка файлова система), в якій можна відформатувати жорсткі диски комп'ютера або створені системними засобами дискові простори. Microsoft готує файлову систему ReFS в якості наступника файлової системи NTFS, технологічні можливості якої вже підійшли до своїх меж. Зокрема, при роботі з носіями даних великого розміру виникають складнощі з їх роботою. Це і занадто тривалий час при виконанні операції перевірки на наявність помилок, і повільна робота журналу, і досягнення обмежень на максимальний розмір файлів.

16.6.1 Особливості файлової системи ReFS

Більшість нововведень ReFS лежать в області створення структур файлів і тек, і управління ними. Ці функції реалізовані з метою автоматичного виправлення помилок, забезпечення високої масштабованості і роботи в режимі Always Online (постійного підключення). Теки у файловій системі ReFS структуровані у вигляді таблиць з файлами в якості записів, які можуть мати власні атрибути, організовані у вигляді підтаблиць. Вільне місце на дисках також організоване в таблицях.

При розробці ReFS переслідувалися такі цілі:

1. Забезпечення максимальної сумісності з існуючими функціями NTFS, і позбавлення від непотрібних функцій, які ускладнюють систему.
2. Верифікація і автоматичне виправлення даних.
3. Масштабованість.
4. Гнучкість архітектури з використанням функції Storage Spaces, яка і була задумана для ReFS.

Функція **Storage Spaces** (дискові простори) – це технологія зберігання даних, призначена для об'єднання в пул надлишкового дискового простору, що дозволяє зменшити ризик втрати даних, забезпечити зручність роботи і зберігання великих об'ємів інформації.

Основні можливості ReFS:

1. Збільшені ліміти на розмір розділів, директорій і файлів.
2. Цілісність метаданих з контрольними сумами.

3. Висока відмовостійкість. Помилки файлової системи, які в NTFS призводили до втрати даних, в ReFS зведені до мінімуму.
4. Автоматичне відновлення вкладених тек і пов'язаних з ними файлів при ушкодженні метаданих.
5. Попереджуюче виправлення помилок. Автоматичне сканування томів на предмет ушкоджень і застосування профілактичних заходів з відновлення даних.
6. Використання надмірного запису для підвищення відмовостійкості.
7. Спеціальна методика запису на диск – Integrity streams, що забезпечує додатковий захист даних при ушкодженні частини диска.
8. Нова модель транзакцій «allocate on write» (copy on write).
9. Disk scrubbing – технологія чищення диска у фоновому режимі.
10. Можливість організації пулів зберігання, які можуть застосовуватися для забезпечення відмовостійкості віртуальних машин і балансування навантаження.
11. Для підвищення продуктивності використовується сегментація послідовних даних (data sriping).
12. Порятунок даних навколо пошкодженої ділянки на диску.

Обмеження файлової системи ReFS наведені в таблиці 16.6.

Таблиця 16.6 – Обмеження файлової системи ReFS

Максимальний розмір файла	– 2^{64}-1 байт
Максимальні розмір тома	– 2^{78} байт при розмірі кластера 16 КБ
Максимальна кількість файлів на томі/в директорії	– 2^{64}
Максимальна довжина імені файла	– 32000 символів Unicode
Максимальна довжина шляху до файла	– 32000
Максимальний розмір будь-якого пулу зберігання	– 4 ПБ
Кількість пулів зберігання в системі	– Не обмежено

ReFS підтримує багато функцій своєї попередниці NTFS, у тому числі:

- шифрування дисків (BitLocker);
- журнал USN;
- списки контролю доступу (ACL);
- символьні посилання для бібліотек;
- точки монтування (mount points);
- точки з'єднання (junction points);
- точки повторної обробки (reparse points).

Усі дані файловій системи ReFS будуть доступні через ті ж самі API, які зараз використовуються для доступу до розділів NTFS.

ReFS була розроблена для того, щоб усунути деякі недоліки файлової системи NTFS, підвищити стійкість, мінімізувати можливі втрати даних, а також працювати з великою кількістю даних.

Одна з головних особливостей файлової системи ReFS – захист від втрати даних. За умовчанням на дисках зберігаються контрольні суми для метаданих або файлів. При операціях читання-запису дані файлів звіряються з контрольними сумами, що зберігаються для них. Таким чином, у разі ушкодження даних є можливість відразу «звернути на це увагу».

У випадку з дисковими просторами її особливості можуть бути найбільш корисними при звичайному використанні. Наприклад, якщо створюється дзеркальні дискові простори з файловою системою ReFS, то при ушкодженні даних на одному з дисків, пошкоджені дані відразу будуть перезаписані з неушкодженої копії іншого диска.

Також нова файлова система утримує інші механізми перевірки, підтримку і виправлення цілісності даних на дисках, причому вони працюють в автоматичному режимі. Для звичайного користувача це означає меншу ймовірність ушкодження даних у випадках, наприклад, раптового відключення живлення при операціях читання-запису.

10.6.2 Відмінності файлової системи ReFS від NTFS

Окрім функцій, пов'язаних з підтримкою цілісності даних на дисках, ReFS має такі основні відмінності від файлової системи NTFS:

1. Вища продуктивність, особливо в разі використання дискових просторів.
2. Теоретичний розмір тому 262144 екзабайти (проти 16 у NTFS).
3. Відсутність обмеження шляху до файлу в 255 символів (у ReFS – 32768 символів).
4. У ReFS не підтримуються імена файлів DOS. Тобто, отримати доступ до теки *C:\Program Files* з шляхом *C:\progra~1* в ній не вийде). У NTFS ця можливість зберігалася з метою сумісності зі старим ПЗ.
5. У ReFS не підтримується стискування, додаткові атрибути, шифрування засобами файлової системи (у NTFS таке є, для ReFS працює шифрування Bitlocker).

Можливо, в майбутньому ReFS може стати основною і єдиною файловою системою в Windows, проте на даний момент цього ще не сталося. Офіційна інформація з файлової системи на сайті Майкрософт [32].

16.7 Файлова система Mac OS

Для операційної системи Mac OS, починаючи з версії 8.1, основною файловою системою є HFS (HFS Plus, Mac OS Extended) [34]. Вона була представлена в 1998 р. і прийшла на зміну HFS (Hierarchical File System), розширивши її можливості. Основні серед них: підтримка імені файлу завдовжки 255 байт з кодуванням UNICODE (точніше, UTF-16 – один із способів кодування символів у системі Unicode) і збільшена 32-бітова адресація. Це дозволяє для

дискових блоків стандартного розміру мати том розміром 2 Тб (4294976296 записів по 512 байт). У період розробки HFS+ мала ім'я Sequoia.

Починаючи з 11 листопада 2002 р., з випуском оновлення 10.2.2, Apple Inc. зробила можливим журналювання для підвищення надійності зберігання інформації. Воно було легко доступне з серверною версією Mac OS X, але тільки через інтерфейс командного рядка з настільних клієнтів.

Як і її попередниця, HFS+ заснована на структурі даних «B-дерево». Останнє є структурою зберігання даних, різновидом дерев пошуку. Такий спосіб був запропонований в 1970 р. Р. Бейером і Е. МакКрейтом [35]. Будучи результатом розвитку бінарних дерев, вони допускають велике число нащадків у будь-якому вузлі (властивість гіллястості). Ще одна їх властивість – збалансованість – полягає в тому, що всі «листя» знаходяться на однаковій відстані від «кореня». Завдяки цьому B-дерева дозволяють швидко знаходити цілі блоки даних за заданим ключом. Заповнення таких структур здійснюється динамічно, в міру заповнення даними. Вони зручні для зберігання великих послідовно оброблюваних блоків даних, а тому часто використовуються в базах даних і файлових системах.

Фізична реалізація B-дерева здійснюється у вигляді мультиоблікової структури сторінок зовнішньої пам'яті, при якій кожному вузлу ієрархії відповідає сторінка (блок зовнішньої пам'яті).

Умови, яким повинно задовольняти B-дерево, такі [34]:

1. Кожна вершина може містити n адресних посилань і $(n-1)$ ключів. Посилання на вершину зліва від ключа забезпечує перехід до вершини дерева з меншими за значенням ключами, а справа – до вершини з великими ключами.
2. Будь-яка некінцева вершина має не менше $n/2$ підпорядкованих вершин.
3. Якщо некінцева вершина містить $k(k \leq n)$ ключів, то їй підпорядкована $(k + 1)$-*а* вершина на наступному рівні ієрархії.
4. Усі кінцеві вершини розташовані на одному рівні.

Серед переваг використання структур типу B-дерев виділимо такі:

- пошук будь-якого запису займає один і той же час; при цьому забезпечується висока продуктивність широкого спектру запитів, наприклад пошуку за заданим значенням або інтервалу;
- ефективність операцій додавання, оновлення і видалення записів, при яких властивість збалансованості підтримується автоматично;
- висока продуктивність як для маленьких, так і для великих наборів даних.

Для ефективної роботи з даними файлова система HFS+ окрім інформаційних блоків містить такі елементи службової інформації (метадані) [36]:

1. *Volume header* (заголовок тому) – містить загальну інформацію про том. Наприклад, розмір блоку даних і інформацію про розташування інших блоків метаданих на диску;

2. *Allocation file (Bitmap)* – файл розміщення, або карта тому, в якому відмічений статус кожного блоку на диску (1 – зайнятий, 0 – вільний);
3. *Catalog file* (каталог) – у ньому зберігається велика частина даних про розміщення файлів і тек на диску;
4. *Extents overflow file* – містить метадані, які не розмістилися в каталозі;
5. *Attributes file* (файл атрибутів) – використовується для контролю доступу і тому подібне;
6. *Journal file* (журнал) – зберігає дані про транзакції, які були виконані для цього тому.

Елементи *Catalog File, Extents Overflow File* **і** *Attribute File* є B-дерева.

Серед інших особливостей даної файлової системи відмітимо те, що окрім традиційно прийнятих атрибутів файлу (системний, обмеження доступу), в HFS+ атрибутом є його ім'я, розмір і навіть сам вміст файлу. Файли в HFS+ можуть мати декілька версій свого вмісту.

Стандартним прийомом збільшення максимального розміру інформації, що зберігається на диску, є зміна розміру мінімальної порції обміну даних – кластера. Він може бути вибраний з такого списку: 512 байт, 2×512 байт, 4×512 байт, 8×512 байт і більше. Стара адресація була серйозним обмеженням HFS, яка не дозволяла працювати з томами обсягом більше 65536 блоків. При об'ємі диска в 1 Гб розмір кластера (блоку) складав 16 Кб, навіть файл з 1 байта займав усі 16 Кб.

Контрольні питання і тести до розділу 16

Контрольні питання

1. Опишіть призначення файлової системи.
2. Що являє собою файл?
3. У чому відмінність каталогу від файлу?
4. Що таке структурований і неструктурований файл?
5. Які два способи може використати файлова система для доступу до логічних записів?
6. Який файл називається індексованим?
7. Що передбачено в індексованому файлі для швидкого пошуку даних?
8. Які одиниці дискового простору використовує ОС при створенні місця під файл на диску?
9. З якої ОС були успадковані обмеження FAT на найменування файлів і каталогів?
10. На які дві області ділиться дисковий простір будь-якого логічного диска у файловій системі FAT?
11. Перелічить переваги файлової системи FAT32 в порівнянні з файловою системою FAT?
12. Перелічить переваги файлової системи NTFS в порівнянні з FAT або FAT32.
13. Які основні особливості має файловоя система NTFS?

14. Які операції виконує файлова система NTFS при збої системи?
15. Що таке «відкладена передача» у файловій системі NTFS?
16. Чи є операція стискування даних обов'язковим атрибутом?
17. До скількох логічних блоків файлу система NTFS застосовує алгоритм стискування?

Тести

1. Файлова система включається до складу ОС для того щоб:
   1) зменшити кількість помилок уведення-виведення;
   2) ефективніше використати дисковий простір;
   3) підвищити продуктивність системи введення-виведення;
   4) забезпечити користувача зручним інтерфейсом для роботи із зовнішньою пам'яттю.
2. Файлова система FAT належить до виду:
   1) перелік блоків;
   2) зв'язаний список блоків;
   3) система з безперервним розміщенням;
   4) зв'язаний список індексів.
3. Мінімальна адресуєма одиниця дискової пам'яті, що виділяється файлу:
   1) сектор;
   2) кластер;
   3) доріжка.
   4) циліндр.
4. У каталозі (теці) можуть зберігатися:
   1) тільки файли;
   2) тільки інші теки;
   3) файли і теки;
   4) файли і вікна Windows.
5. Файл – це:
   1) область зберігання даних на диску;
   2) програми або дані, що зберігаються в довготривалій пам'яті;
   3) програми або дані, що мають ім'я і зберігаються в оперативній пам'яті;
   4) програми або дані, що мають ім'я і зберігаються в довготривалій пам'яті.
6. Таблиці FAT і MFT потрібні для:
   1) зберігання інформації про носій;
   2) резервного копіювання даних;
   3) зберігання секторів в логічному дисковому адресному просторі;
   4) зберігання інформації про файли, що зберігаються.
7. При видаленні файлу на носії даних:
   1) автоматично відбувається форматування носія;
   2) місце, яке займав файл, позначається як вільне;

3) автоматично відбувається стискування носія;
4) носій вимагає форматування.

8. Журналізація у файлових системах застосовується для:
   1) протоколювання дій користувачів;
   2) підвищення відмовостійкості системи;
   3) можливості відміняти помилкові зміни даних у файлах користувачів.
9. Використання блоку диска розміром 8бКб в порівнянні з блоком розміром 4 Кб вигідніше, оскільки:
   1) у цьому блоці поміщається більше сторінок пам'яті;
   2) у цьому блоці можна розмістити більше файлів;
   3) обмін з диском здійснюється швидше.
10. Основною перевагою використання таблиці відображення файлів (FAT) в порівнянні з класичною схемою виділення зв'язним списком є:
    1) скорочення кількості звернень до диску;
    2) підвищена надійність;
    3) економічне використання дискового простору.
11. Основним недоліком фізичної організації файлу у вигляді безперервного розміщення є:
    1) низька швидкість доступу;
    2) фрагментація;
    3) великі витрати на пошук і зчитування кластерів;
    4) дефрагментація.
12. Багато ОС підтримують імена файлів, що складаються з двох частин (ім'я+розширення). Це робиться для того, щоб:
    1) спростити запам'ятовування імені файлу;
    2) спростити сортування файлів при виведенні списку файлів в каталозі;
    3) операційна система могла зв'язати це ім'я із прикладною програмою, яка повинна обробляти цей файл.
13. Розмір кластера:
    1) дорівнює розміру сектора;
    2) менше розміру сектора;
    3) завжди більше розміру сектора;
    4) кратний розміру сектора.

17. ВІДПОВІДІ НА КОНТРОЛЬНІ ПИТАННЯ І ТЕСТИ

17.1 Відповіді на контрольні питання

Розділ 1.

1. ОС – це комплекс програм, контролюючих роботу прикладних програм і системних додатків, які виконують роль інтерфейсу між користувачами, прикладними програмами, системними додатками і апаратним забезпеченням комп'ютера.

4. Важливим плюсом ОС 3-го покоління стала здатність зчитувати завдання з перфокарт на диск. Потім, як тільки закінчувалося поточне завдання, з диску (стрічки) зчитувалося наступне завдання. Цей технічний прийом називається «***підкочуванням»*** даних або спулінгом. Коли одне із завдань чекає завершення операцій введення-виведення, процесор перемикається на інше завдання. Такий режим відомий як ***багатозадачність***.

5. Варіант мультипрограмних систем – системи ***розподілу часу***, варіант багатозадачності, при якому в кожного користувача є свій термінал.

6. Перша система з режимом розподілу часу для управління обчислювальними ресурсами комп'ютера IBM 7090 була розроблена на початку 60-х років групою програмістів під керівництвом професора Фернандо Корбато.

7. Уперше ***віртуальну пам'ять*** – метод управління пам'яттю комп'ютера, що дозволяє виконувати програми, які вимагають більше оперативної пам'яті, чим є в комп'ютері – почали застосовувати в ОС MULTICS (1965 р.).

11. Початок 80-х років пов'язаний зі знаменними для історії операційних систем подіями – появою Великих Інтегральних Схем і ***персональних комп'ютерів***.

12. У 1974 році Гарі Килдолл створив першу дискову операційну систему, названу **CP/M** (Control Program for Microcomputers).

14. У 60-і роки **Даг Энгельбарт (Doug Engelbart),** винайшов ***графічний інтерфейс користувача*** (**GUI**, Graphical User Interface), що складається з вікон, значків, різних меню і миші.

17. В університеті Карнегі-Меллона був розроблений проект **Mach –** мікроядерна архітектура ОС (1985-1994 рр.). Проектом керував **Річард Ф. Рашид**. Мікроядро ОС Mach управляло процесами, обміном повідомлень між ними, віртуальною пам'яттю і драйверами пристроїв. Інша частина ОС реалізовувалася у вигляді серверів – програм, які виконувалися в режимі користувача.

21. У перше десятиліття після своєї появи більшість смартфонів працювали під управлінням Symbian OS.

Розділ 2.

3. Завантаження в оперативну пам'ять програм, розподіл ресурсів між програмами, прийом і виконання запитів від програм, що виконуються, організація спрощеного доступу користувача до ресурсів обчислювальної системи.

6. Щоб процес міг бути виконаний, ОС повинна призначити йому область оперативної пам'яті, в якій будуть розміщені коди і дані процесу, а також надати йому необхідну кількість процесорного часу, доступ до таких ресурсів, як файли і пристрої введення-виведення.

7. Сукупність усіх областей оперативної пам'яті, виділених операційною системою процесу, називається його ***адресним простором***.

8. Стан операційного середовища ідентифікується станом регістрів і програмного лічильника, режимом роботи процесора, покажчиками на відкриті файли, інформацією про незавершені операції введення-виведення, кодами помилок, системних викликів, що виконуються цим процесом тощо. Ця інформація називається ***контекстом процесу***.

14. При невитісняючій багатозадачності активний процес виконується до тих пір, поки він сам не віддасть управління ОС для того, щоб та вибрала з черги інший готовий до виконання процес. При витісняючій багатозадачності рішення про перемикання процесора з одного процесу на інший приймається ОС, а не активним процесом.

16. Ні. Операційні системи управляють також додатками і іншими програмними об'єктами, такими як віртуальні машини.

17. Ні. Багато ОС надає мінімальний інтерфейс користувача. Основними задачами ОС є забезпечення взаємодії між додатками і апаратними засобами комп'ютера, а також розподіл програмних і апаратних ресурсів системи.

Розділ 3.

4. Розрізняють два рівні сумісності: на рівні двійкової ***сумісності*** і сумісності на ***рівні початкових текстів додатків***.

5. Роботи по стандартизації інтерфейсу ОС відбуваються в рамок проекту POSIX (Portable Operating System Interfice – Переносимий Інтерфейс ОС), який триває і в наші дні.

12. Апаратура комп'ютера підтримує як мінімум два режими роботи – ***режим користувача*** і ***привілейований режим,*** який також називають ***режимом ядра,*** або ***режимом супервізора***.

Розділ 4.

6. BIOS містить драйвери для усіх пристроїв, що входять у базову конфігурацію комп'ютера: жорстких і гнучких дисків, клавіатури, дисплея тощо.

8. Регістри, доступні користувачеві. До цих регістрів користувач може звертатися за допомогою команд машинної мови. Зазвичай серед доступних регістрів є регістри даних, адресні регістри і регістри коду умови.

11. Якщо дані виявляються в кеш-пам'яті, тобто сталося ***кеш-попадання,*** то вони зчитуються з неї, і результат передається джерелу запиту. Якщо потрібних даних немає в кеш-пам'яті, тобто стався ***кеш-промах***, то вони разом зі своєю адресою копіюються з оперативної пам'яті в кеш-пам'ять.

13. У більшості реалізацій кеш-пам'яті відсоток кеш-попадань виявляється дуже високим – понад 90%. Таке високе значення ймовірності знаходження

даних в кеш-пам'яті пояснюється наявністю у цих об'єктивних властивостей: просторовій і тимчасовій локальності.

15. Виключення повідомляють про помилки (наприклад, ділення на нуль) і звертаються до ОС, щоб та визначила, як реагувати на це. ОС може відповісти бездіяльністю або завершити процес.

Розділ 5.

2. Ні. Процесом називається програма в стадії виконання, тоді як сама програма є неживим логічним об'єктом.

3. Термін «процес» уперше почали застосовувати розробники операційної системи MULTICS в 60-х роках.

13. Причиною введення додаткового стану призупиненого (блокованого) процесу було повільне, в порівнянні з обчисленнями, виконання операції введення-виведення, яке призводило до простоїв CPU в однозадачній системі.

14. Якщо в основній пам'яті немає жодного готового до виконання процесу, ОС переводить один з блокованих процесів на диск (здійснює його свопінг), розміщуючи його в чергу ***призупинених* (*блокованих*) *процесів***, які тимчасово витиснені з основної пам'яті. Далі ОС завантажує інший процес з черги призупинених, після чого продовжує його виконання.

16. Оскільки в задачі ОС входить управління процесами і ресурсами, вона повинна мати в розпорядженні інформацію про поточний стан кожного процесу і ресурсу. Універсальний підхід до надання такої інформації простий: ОС створює і підтримує таблиці з інформацією для кожного об'єкту управління.

Розділ 6.

1. В ОС, разом з більшою одиницею роботи (процесами), потрібний інший механізм розпаралелювання обчислень, який враховував би тісні зв'язки між окремими гілками обчислень одного і того ж додатку, і вимагав для свого виконання дещо дрібніших робіт. Для цих цілей сучасні ОС пропонують механізм багатопотокової обробки – ***потік виконання.***

7. Подібно до традиційних процесів, потоки можуть породжувати потоки-нащадки, можуть переходити із стану в стан (ВИКОНАННЯ, ОЧІКУВАННЯ і ГОТОВНІСТЬ).

10. Щоб додаток був багатопотоковим його потрібно створити з використанням спеціальної бібліотеки, яка є пакетом програм для роботи з потоками на рівні ядра. Така бібліотека містить код, який дозволяє створювати і видаляти потоки, здійснювати обмін повідомленнями і даними між потоками, планувати їх виконання, а також зберігати і відновлювати їх контекст.

17. Під ***асинхронними*** подіями розуміють події, які відбуваються незалежно (можливо одночасно) за винятком випадків, коли залежність встановлюється зовнішніми силами.

18. У відсутності стандартних бібліотек для роботи з потоками, які були б реалізовані на усіх платформах.

19. Потоки, що належать одному процесу, можуть обмінюватися один з одним даними за допомогою загального адресного простору, обходячись без механізму взаємодії процесів за допомогою ядра.

20. Потоки рівня користувача представляють для додатка власний API, який не залежить від API ОС.

Розділ 7.

1. Так. Якщо два процеси бажають отримати доступ до одного ресурсу, то ОС виділить цей ресурс одному з процесів, тоді як другий процес вимушений чекати на завершення роботи з ресурсом першого процеса.

2. Результат виконання потоків залежить від послідовності виконання операцій потоків в системі. Усе це призводить до того, що в одній ситуації код може виконуватися правильно, а в іншій неправильно.

3. Такі ситуації, в яких два (і більше) потоки (процеси) зчитують або записують дані одночасно, і кінцевий результат залежить від того, який з них був першим, тобто від співвідношення швидкостей потоків, називають ***станом перегонів,*** або ***змагань*** (race condition).

5. Цей прийом називають ***взаємним виключенням*** або ***блокуванням***.

8. Алгоритм ***Лемпорта*** ще називають «алгоритм булочної», який розв'язує задачу взаємного виключення для *N* процесів як у багатопроцесорних, так і в розподілених систем обробки даних.

10. Якщо потік не покине свою критичну ділянку.

Розділ 8.

3. В окремому випадку, коли семафор *S* може набувати тільки значень 0 і 1, то така спрощена версія семафора, називається ***мьютексом*** (mutex, скорочення від **mut**ual **ex**clusion).

4. Якщо для зберігання процесів використовується черга, то такий семафор називається ***сильним семафором.*** Семафор, порядок витягання процесів з черги якого не визначений, називається ***слабким семафором.***

9. Якщо процеси виконуються на різних машинах, то такі механізми як семафори і монітори тут не застосовуються. Одним з підходів до забезпечення синхронізації процесів є ***передача повідомлень***.

11. Непряма адресація припускає, що повідомлення посилаються не прямо від відправника одержувачеві, а посилаються в спільно використовувану структуру даних, що складається з черг для тимчасового зберігання повідомлень, які називають ***поштовими скриньками*** (mailbox).

13. Так. При розміщенні в чергу очікування семафора потік блокується, будучи не в змозі виконати програмний код.

14. Виклик операції **V** приведе до того, що значення семафора збільшиться, це може призвести до того, що більш ніж один потік може увійти до своїх критичних ділянок.

15. Потік з низьким пріоритетом може опинитися в ситуації нескінченного відкладання

Розділ 9.

7. Можливість відібрати ресурс у процесу, дозволити використати його іншому процесу, а потім повернути його без сповіщення процесу багато в чому залежить від природи цього ресурсу. Відновлення цим способом частенько ускладнене або зовсім неможливе.

10. Можна уникнути взаємоблокування, якщо розподіляти ресурси, дотримуючись певних правил. Серед такого роду алгоритмів найбільш відомий «алгоритм банкіра», який базується на ***безпечних*** або ***надійних*** станах (safe state).

12. Один із способів атаки умови кругового очікування – діяти відповідно до правила, згідно з яким кожен процес може мати тільки один ресурс в кожен момент часу. Якщо потрібний другий ресурс – звільни перший. Інший спосіб – упорядкувати ресурси – і замовляти їх в певному порядку.

13. Взаємоблокування і нескінченне відкладання схожі в тому, що і ті, і інші виникають тоді, коли процес чекає певної події.

14. Будь-який процес перед початком роботи повинен вказати максимальне число ресурсів, які знадобляться йому під час роботи.

15. Ні. Якщо процеси звільнятимуть займані ними ресурси, то система може перейти з ненадійного в надійний стан.

16. Тому що комп'ютери, що входять до складу розподіленої системи, можуть управлятися різними ОС.

Розділ 10.

10. Спираючись на емпіричні результати, можна відмітити, що основний об'єм пам'яті, яку займає процес, велику частину часу залишається вільним, то існує таке правило ***"дев'яносто до десяти",*** або ***правило локалізації (локальності),*** яке стверджує, що ***90% звернень до пам'яті в процесі доводиться на 10 % його адресного простору***.

14. При завантаженні процесу частина його віртуальних сторінок поміщається в оперативну пам'ять, а інші – на диск. Частина процесу, розташована в деякий момент часу в основній пам'яті, називається ***резидентною множиною*** процесу.

Розділ 11.

2. Ефективність алгоритму зазвичай на конкретній послідовності посилань до пам'яті, для якої підраховується число виникаючих ***сторінкових порушень*** (page faults). Ця послідовність називається ***рядком звернень*** (reference string).

3. Реалізувати оптимальний алгоритм неможливо, оскільки для цього системі необхідно знати усі майбутні події. ОС не знає, до якої сторінки буде наступне звернення.

5. У простій ***схемі годинної стратегії*** з кожним кадром зв'язується один додатковий біт, відомий як біт використання (***u - use,*** або біт звернення – ***r - referenced***).

6. Підвищити ефективність годинникового алгоритму можна шляхом збільшення кількості використовуваних при його роботі бітів (біт модифікації).

Такий алгоритм називають ***модифікованим годинниковим алгоритмом*** або ***алгоритмом, що не використовує останнім часом сторінки***.

8. Виходячи з евристичного правила, що недавнє минуле – хороший орієнтир для прогнозування найближчого майбутнього – має сенс заміщати сторінку, яка не використовувалася впродовж найдовшого часу. Такий підхід називається ***Least Recently Used алгоритм (LRU)*** – сторінка, що не використалася найдовше. Згідно з принципом локалізації можна чекати, що ця сторінка не використовуватиметься і в найближчому майбутньому.

9. Для цього алгоритму потрібний програмний лічильник, пов'язаний з кожною сторінкою в пам'яті. За допомогою лічильника намагаються відстежити, як часто відбувалося звернення до кожної сторінки. При сторінковому перериванні для заміщення вибирається сторінка з найменшим значенням лічильника.

Розділ 12.

4. Короткостроковий планувальник, відомий також як ***диспетчер***, працює найчастіше, визначаючи, який саме процес виконуватиметься наступним. Вибір нового процесу для виконання робить вплив на функціонування системи до настання чергової аналогічної події.

5. Розрізняють дві основні стратегії планування – ***витісняюча*** і ***невитісняюча багатозадачність***. При витісняючій багатозадачності процеси, які виконуються, можуть бути перервані планувальником ОС. При невитісняючій багатозадачності процес може виконуватися впродовж необмеженого часу і не може бути перерваний ОС.

8. В основі багатьох витісняючих алгоритмів планування лежить концепція квантування. Відповідно до цієї концепції кожному потоку по черзі для виконання надається обмежений безперервний період процесорного часу – квант.

12. Іншою важливою концепцією, що лежить в основі багатьох витісняючих алгоритмів планування, є пріоритетне обслуговування. Пріоритетне обслуговування припускає наявність у потоків деякої спочатку відомої характеристики – пріоритету, на підставі якого визначається порядок їх виконання. В цьому випадку є ризик «голодування» довгих процесів.

19. Основний ризик при використанні стратегії SPN полягає в можливому «голодуванні» довгих процесів при стабільній роботі коротких процесів. Його застосування небажане в системах з розподілом часу або в системах обробки транзакцій через відсутність витіснення.

20. Стратегія найменшого часу, що залишається (shortest remaining time – **SRT**), є витісняючою версією стратегії SPN. В цьому випадку планувальник вибирає процес з найменшим очікуваним часом до закінчення процесу. При приєднанні нового процесу до черги готових для виконання процесів може виявитися, що його час менший, ніж час, що залишився, виконуваного процесу. Планувальник, відповідно, може застосувати витіснення при готовності нового процесу.

22. Завдяки дисципліні SPN зменшується середній час очікування процесів, що дозволяє підвищити пропускну спроможність системи.

23. Ні. Чим довше чекає процес, тим вище ймовірність, що він буде запущений раніше, ніж короткий процес.

24. Ні. Це визначає диспетчер пам'яті. Планувальник процесів розподіляє між процесами процесорний час.

Розділ 13.

4. У разі використання архітектури «головний-підлеглий» ключові функції ОС завжди виконуються на одному спеціально виділеному процесорі. Усі інші процесори можуть виконувати тільки додатки користувача.

5. При використанні архітектури рівноправних процесорів ОС може виконуватися на будь-якому з процесорів, і кожен процесор самостійно планує свою роботу, беручи процеси для виконання із загального пулу.

Розділ 15.

1. **Блок-орієнтовані пристрої** зберігають інформацію в блоках фіксованого розміру, кожен з яких має свою власну адресу, і виконують передачу даних поблочно (наприклад, диск). Оскільки **байт-орієнтовані пристрої** не адресуються і не дозволяють робити операцію пошуку, вони генерують або споживають послідовність байтів (наприклад, клавіатура, миша, термінали).

5. Цінною властивістю ОС є можливість динамічно завантажувати в оперативну пам'ять необхідний драйвер (без зупинки ОС), і вивантажувати його після того, як потреба в підтримці пристрою минула, що може істотно заощадити системну область пам'яті.

6. Синхронний режим означає, що програмний модуль призупиняє свою роботу до тих пір, поки операція введення-виведення не буде завершена, а при асинхронному режимі програмний модуль продовжує виконуватися в мультипрограмному режимі одночасно з операцією введення-виведення.

7. Щоб зменшити накладні витрати і збільшити ефективність, іноді зручно виконувати зчитування даних заздалегідь, до реального запиту, а запис даних – трохи пізніше за реальний запит. Ця методика відома як ***буферизація***.

10. Для звільнення процесора від операцій послідовного виведення даних з оперативної пам'яті або послідовного введення в неї був запропонований механізм прямого доступу зовнішніх пристроїв до пам'яті – ПДП або Direct Memory Access – **DMA**.

Розділ 16.

3. ***Каталог*** – це, з одного боку, група файлів, об'єднаних користувачем виходячи з деяких міркувань (наприклад, файли, що містять програми ігор), а з іншого боку – це файл, що містить системну інформацію про групу файлів, його складових.

5. Файлова система може використати два способи доступу до логічних записів: читати або записувати логічні записи послідовно (***послідовний доступ)***, або позиціонувати файл на запис з вказаним номером ***(прямий доступ***).

6. Файли, доступ до записів яких здійснюється послідовно, за номерами позицій, називаються ***неіндексованими,*** або ***послідовними.*** Іншим типом файлів є ***індексовані файли***, вони допускають швидший прямий доступ до окремого логічного запису. В індексованому файлі записи мають одне або більше ключових (індексних) полів і можуть адресуватися шляхом вказівки значень цих полів.

7. Для швидкого пошуку даних в індексованому файлі передбачається спеціальна ***індексна таблиця***, в якій значенням ключових полів ставиться у відповідність адреса зовнішньої пам'яті. Ця адреса може вказувати або безпосередньо на шуканий запис, або на деяку область зовнішньої пам'яті, займану декількома записами, до числа яких входить шуканий запис.

8. Операційна система при роботі з диском використовує, як правило, власну одиницю дискового простору, що називається ***кластером*** (cluster). Іноді кластер називають ***блоком*** (наприклад, в ОС Unix).

9. Обмеження FAT на найменування файлів і каталогів успадковані з ОС CP/M. Нова видозмінена система, перейменована в MS-DOS (Microsoft Disk Operation System), майже повністю успадкувала структуру FAT від своєї попередниці.

10. У файловій системі FAT дисковий простір будь-якого логічного диску ділиться на дві області: ***системну область*** і ***область даних***. Системна область логічного диску створюється та ініціалізується при форматуванні, а згодом оновлюється при маніпулюванні файловою структурою. Область даних містить файли і каталоги, які підлеглі кореневому каталогу.

14. При збої системи NTFS виконує три проходи: аналізу, повторів і відкатів. В процесі аналізу на підставі інформації файлу реєстрації NTFS оцінює ушкодження і точно визначає, які кластери треба модифікувати. Під час повторного проходу виконуються усі етапи транзакції від останньої контрольної точки. При відкаті відбувається повернення усіх незавершених транзакцій.

17.2 Відповіді на тести

Розділ 1.

1. 3) UNIX.
2. 2) IBM/360.
3. 1) як графічна оболонка, командою win в системі MS-DOS.
4. 4) електронні лампи.
5. 4) Лінус Торвальдс.
6. 2) систем розподілу часу.
7. 3) лампи.
8. 1) системи пакетної обробки.
9. 3) з появою попереднього запису пакету завдань на магнітний диск.
10. 2) тому що програми користувача мають право заборони переривань.
11. 1) в ОС Alto для комп'ютера Alto (компанія Xerox).
12. 3) сукупність стандартів, які використовуються в ОС Unix.

Розділ 2.

1. 3) операційних середовищ. Операційна система виконує функції управління процесами в обчислювальній системі, розподіляє ресурси обчислювальної системи між різними процесами і утворює програмне середовище, в якому виконуються програми користувача. Таке середовище називається операційним.

2. 1) системних ресурсів комп'ютера. Практично всі ОС мають API (безліч системних викликів), за допомогою якого програмісти можуть створювати додатки для цієї операційної системи.

3. 2) системного програмного забезпечення.

4. 4) система, що надає користувачеві інтерфейс до ПК і управляє його ресурсами.

5. 1) забезпечити зручність, ефективність, надійність і безпеку використання комп'ютерного устаткування, зовнішніх пристроїв і призначених для користувача програм.

6. 3) операційною системою.

7. 2) сервери.

Розділ 3.

1. 1) діалогового режиму роботи і режиму мультипрограмування.
2. 3) у привілейованому режимі.
3. 1) менш продуктивна.
4. 2) постійно знаходяться в оперативній пам'яті.
5. 4) ядро ОС.
6. 4) легше.
7. 3) режимі користувача.
8. 3) розширюваність, надійність, переносимість.
9. 3) 4.
10. 2) розширюваною.
11. 1) мікроядерної архітектури.
12. 1) модулі ядра.
13. 2) спрощує.
14. 2) ні.
15. 4) режимі користувача.
16. 2) двома перемиканнями режиму (привілейований/користувача).
17. 2) двома перемиканнями режиму (привілейований/користувача).
18. 1) захищеність.
19. 2) мобільність.
20. 4) розширюваність.
21. 3) масштабованість.
22. 4) замість двох перемикань режиму процесора в разі системного виклику здійснюється чотири (два – під час обміну між клієнтом і мікроядром, два – між сервером і мікроядром).
23. 4) обміну повідомленнями, що передаються через мікроядро.

Розділ 4.

1. 4) регістр, кеш-пам'ять, оперативна пам'ять, жорсткий диск, оптичний диск.

2. Ні. BIOS записується в постійний пристрій пам'яті до інсталяції на комп'ютер ОС і може забезпечувати роботу різних ОС.

3. Так. Розрядність шини визначає об'єм даних, яким можуть обмінюватися основна пам'ять і процесор за один такт. Якщо процесор генерує запити на більшу кількість даних, чим можна передати за один такт, то це призводить до затримки роботи процесора.

4. 3) швидкодіючу статичну пам'ять.

5. 1) буфери в оперативній пам'яті, в яких осідають найактивніше використовувані дані.

Розділ 5.

1. 2) дескриптора.

2. 1) відібрані в процесу. Вивантажений ресурс – це ресурс, який безболісно можна забрати в процесу (наприклад, пам'ять). Невивантажений ресурс – це ресурс, який не можна забрати в процесу без втрати даних (наприклад, принтер).

3. 2) інформація, необхідна для вирішення задачі планування.
4. 1) із стану «виконання».
5. 4) очікування.
6. 2) із стану «готовність».
7. 4) контекст.
8. 3) синхронізація.
9. 2) режим роботи процесора.
10. 3) очікування → виконання.
11. 2) покажчики на ресурси, якими управляє процес.
12. 3) очікування події.
13. 4) сокет.

Розділ 6.

1. 4) програмний лічильник, стек і вміст регістрів.
2. 3) після завершення процесу-батька.
3. 2) синхронізація.
4. 2) очікування завершення введення-виведення або іншої події.
5. 2) готовності.
6. 4) потік.
7. 3) готовності.
8. 2) хоч би один потік процесу знаходиться в стані готовність.
9. 4) готовність.
10. 4) потоку.
11. 2) 2,3,5.
12. 3) Лічильник команд, Регістри, Стек, Стан.
13. 1) 1, 3, 4.

14. 2) 2, 4.

15. 4) ОС нічого не знає про потоки на рівні користувача, для неї відомий тільки один головний потік (процес), системний виклик якого вона і обробляє.

16. 2) так, процес знаходиться в стані «закінчив виконання», якщо усі його потоки знаходяться в стані «закінчив виконання».

17. 4) процес може містити декілька потоків.

18. Ні. Потоки, що належать одному процесу, можуть обмінюватися один з одним даними за допомогою загального адресного простору, обходячись без механізму взаємодії процесів, що має на увазі звернення до ядра.

19. Так. Більшість додатків містять фрагменти коду, які можуть виконуватися незалежно від іншої частини додатка. Якщо виділити ці фрагменти в окремі потоки, можна буде виконувати їх окремо на різних процесорах/ядрах.

Розділ 7.

1. 1) синхронізації.

2. 3) неподільна операція, що виконується без переривання діяльності.

3. 3) до ділянки коду процесу, виконання якого спільне з іншими процесами може призвести до неоднозначних результатів.

4. 2) пристрій або дані, до яких процес має ексклюзивний доступ.

5. 1) до набору процесів, що спільно використовують який-небудь ресурс.

6. 4) ресурсом.

Розділ 8.

1. 2) монітори.

2. 1) станеться взаємне блокування процесів.

3. 3) 3.

4. 3) взаємовиключення гарантується всередині монітором за допомогою програмних алгоритмів, або апаратних засобів, або семафорів, розглянутих раніше до моніторів.

5. 2) якби потік чекав звільнення ресурсу всередині монітора, то жоден інший потік не зміг би увійти до монітора, щоб повернути ресурс системі.

6. 4) семафора.

7. 2) в адресному просторі ядра операційної системи.

8. Так. При розміщенні в чергу очікування семафора потік блокується, будучи не в змозі виконати програмний код, із-за якого він міг би опинитися в черзі очікування іншого семафора.

9. Ні. Монітор може мати окремі змінні-умови для кожної окремої ситуації, яка може привести до виклику команди очікування в моніторі.

Розділ 9.

1. 4) запобігання тупику.

2. 3) виявлення тупика.

3. 1) виконання усіх перерахованих умов.

4. 2) ігнорування проблеми в цілому.

5. 4) умову взаємного виключення.

6. 1) надійним.

7. 1) виникнення тупика.

8. 4) алгоритм банкіра призначений для роботи тільки з постійним числом системних ресурсів.

9. 2) так, оскільки ОС завжди може забрати цей ресурс у результаті закінчення кванта часу або призупинення процесу і виділити його іншому процесу, а потім повернути його назад.

10. 3) умову невивантажуваності (відсутності перерозподілу) – у процесу не можна примусово забрати раніше отримані ресурси. Процес, що володіє ними, повинен сам звільнити ресурси.

11. 2) ненадійним.

12. 3) організувати спулінг.

13. 3) система з розподілом часу.

14. 1) безпечна.

Розділ 10.

1. 4) сегментів між оперативною і зовнішньою пам'яттю.

2. 4) уповільнення виконання.

3. 3) завантажувати певну кількість програм, розмір яких перевищує об'єм доступної фізичної пам'яті.

4. 2) спрощує компонування. Окрім простоти управління структурами даних, що збільшуються або скорочуються, сегментована пам'ять має і інші переваги. До них відносяться: простота компонування окремо скомпільованих процедур.

5. 1) у спеціальній швидкій пам'яті процесора і в оперативній пам'яті.

6. 1) отримання великого адресного простору без придбання додаткової фізичної пам'яті.

7. 3) структура, що використовується для відображення логічного адресного простору у фізичний при сторінковій організації пам'яті.

8. 2) розрядністю процесора.

9. 4) 7 і 0.

10. 3) віртуальна пам'ять.

11. 4) деякі послідовності звернень до сторінок призводять до збільшення числа сторінкових порушень при збільшенні виділених процесу кадрів.

12. 3) зменшити об'єм пам'яті, що витрачається на відображення віртуального адресного простору у фізичний.

13. 2) 2^{20}.

14. 1) структура, яка використовується для відображення логічного адресного простору у фізичний при сторінковій організації пам'яті.

15. 3) втрата частини пам'яті, яка не виділена жодному процесу.

16. 4) сукупність програмно-апаратних засобів, що дозволяють користувачам писати програми, розмір яких перевершує наявну оперативну пам'ять.

17. 4) динамічними розділами.

18. 3) сегмент фізичної пам'яті, відображений у віртуальний адресний простір декількох процесів.

19. 3) спеціальні регістри процесора.

20. 2) структура, яка використовується для відображення логічного (віртуального) адресного простору у фізичний при сторінковій організації пам'яті.

21. 1) зовнішньої фрагментації.

22. 4) значна кількість часу, що витрачається на процедуру стискування.

23. 1) сторінками фіксованого розміру.

24. 2) образи процесів вивантажуються на диск і повертаються в оперативну пам'ять цілком.

25. 2) сторінкова організація.

26. 1) втрата частини пам'яті, не виділеної жодному процесу.

27. 3) так, якщо розмір сторінки дорівнює 512 байт.

28. 4) два рівні.

29.3) порівняння адреси з усіма записами кеша одночасно.

30. 4) 4 Кбт. Для обчислення P необхідно S*R/P + P/2=0 і знайти похідну цієї функції по P.

31. 2) так, якщо розмір сторінки дорівнює 1 Кб (або 512 байт).

32. 1) динамічний розподіл пам'яті переміщуваними розділами.

Розділ 11.

1. 4) сторінку-кандидат на видалення з пам'яті і зберегти сторінку, що видаляється, на диску, якщо вона зазнала зміни.

2. 1) практично скрутна. LRU – хороший, але важкий в реалізації алгоритм. Необхідно мати зв'язаний список усіх сторінок в пам'яті, на початку якого будуть зберігатися нещодавно використані сторінки. Причому цей список повинен оновлюватися при кожному зверненні до пам'яті. Багато часу треба і на пошук сторінок в такому списку.

3. 2) 3:

4. 3) 6.

5. 3) 4.

6. 2) 5.

7. 2) FIFO.

8. 1) LRU.

9. 3) OPT.

Розділ 12.

1. 1) частоті виконання. Планування завдань використовується як довгострокове планування процесів. Воно відповідає за породження нових процесів в системі, визначаючи її міру мультипрограмування, тобто кількість процесів, що одночасно знаходяться в ній. Якщо міра мультипрограмування системи підтримується постійною, тобто середня кількість процесів в комп'ютері не міняється, то нові процеси можуть з'являтися тільки після завершення раніше завантажених. Тому довгострокове планування здійснюється досить рідко.

2. 4) динамічним.

3. 1) вибір чергового завдання для запуску і виділення йому кванта часу.

4. 3) довгострокове планування.

5. 2) середньострокове планування.

6. 1) короткострокове планування.

7. 4) First Come, First Served (FCFS).

8. 2) Round Robin (RR).

9. 3) Shortest Process Next (SPN).

10. 1) пріоритетне планування.

11. 4) T > 30.

12. 3) First Come, First Served (FCFS).

13. 1) 2.3.

14. 3) 5.

15. 2) 2.

16. 3) 5.3.

17. 4) 8.

18. 2) 4.3.

19. 2) 2.6.

20. 4) 4.6.

21. 3) Shortest Process Next (SPN).

22. 1) в системах з абсолютними пріоритетами виконання активного потоку переривається, якщо в черзі готових потоків з'явився потік, що має більший пріоритет.

23. 2) в системах з відносними пріоритетами виконання активного потоку триває до тих пір, поки він сам не покине процесор.

24. 2) менше.

25. 4) витісняючою.

26. 2) активний потік виконується до тих пір, поки він сам не віддасть управління ОС.

27. 1) не переривається.

28. 1) коротким задачам.

29. 3) довгим задачам.

30. 1) P1.

31. 4) короткострокового планування (планування нижнього рівня).

32. 1) середньостроковий планувальник.

33. 1) квантування і пріоритети.

34. 3) 3.

35. 4) резервний.

36. 1) абсолютних пріоритетів.

Розділ 13.

1. 2) мультипроцесорна обробка.

2. 1) слабкозв'язану систему.

3. 2) сильнозв'язану систему.

4. 3) розподілену систему.

5. 2) ініціатором виконання роботи мережевою службою завжди виступає клієнт, а сервер завжди знаходиться в режимі пасивного очікування запитів.

6. 2) симетричні.

7. 2) всі члени бригади запускаються одночасно на різних центральних процесорах.

Розділ 14.

1. 1) служить для управління яким-небудь об'єктом в реальному масштабі часу.

2. 4) у системах реального часу.

3. 1) не під час роботи системи, а заздалегідь.

4. 2) мультипроцесорна обробка.

5. 1) реактивністю.

6. 1) «абсолютний» пріоритет.

7. 2) до «жорсткої» системи реального часу.

Розділ 15.

1. 4) асинхронний.

2. 2) буферизації.

3. 4) управляють пристроями введення-виведення, прийомом і передачею даних через порти і виставлянням сигналів на магістралі.

4. 1) клавіатура, миша, принтер, послідовний порт.

5. 3) магнітний диск, оптичний диск.

6. 4) електронний компонент для управління зовнішнім пристроєм і організації обміну даними.

7. 1) призупинення програми, яка дала запит на передачу.

8. 2) введення-виведення, при виконанні якого програма призупиняється і чекає його закінчення.

9. 4) розрядність (розмір) елементу в таблиці FAT.

10. 3) FCFS.

11. 2) SSTF.

12. 1) SCAN.

13. 4) C-SCAN.

14. 1) SCAN.

15. 2) SSTF.

16. 4) C-SCAN.

17. 3) драйвером.

18. 4) номер головки, номер циліндра, номер сектора.

19. 3) чим ближче доріжка до центру.

20. 3) сектор.

21. 4) 0 від зовнішнього краю до центру.

22. 3) 38 мс.

23. 1) 30 мс.

24. 2) 50.

25. 3) 32.

26. 4) 26.
27. 2) 20.

Розділ 16.

1. 4) забезпечити користувача зручним інтерфейсом для роботи із зовнішньою пам'яттю.

2. 2) зв'язаний список блоків.

3. 2) кластер.

4. 3) файли і теки.

5. 4) програми або дані, що мають ім'я і зберігаються в довготривалій пам'яті.

6. 4) зберігання інформації про файли, що зберігаються.

7. 2) місце, яке займав файл, позначається як вільне.

8. 2) підвищення відмовостійкості системи.

9. 3) обмін з диском здійснюється швидше.

10. 1) скорочення кількості звернень до диску.

11. 2) фрагментація.

12. 3) операційна система могла зв'язати це ім'я із прикладною програмою, яка повинна обробляти цей файл.

13. 4) кратний розміру сектора.

СПИСОК ЛІТЕРАТУРИ

1. Робби А., Престон Г. Windows XP. Сборник рецептов. – СПб.: Питер, 2007. – 656 с.
2. Амарис К, Драуби О., Мистри Р., Моримото Р., Ноэл М. Windows Server 2008 R2. Полное руководство. – М.: Вильямс, 2010. – 1456 с.
3. Сафонов В.О. Основы современных операционных систем. – М.: Национальный открытый университет «ИНТУИТ», 2016. – 868 с.
4. Тюрин В.Ф., Зельдинова С.А., Крайнева И.А. Операционная система ДИСПАК. Электронный ресурс (на 14.12.2016): http://www.computer-museum.ru/articles/operatsionnye-sistemy/789/
5. Авраменко В.С., Салапатов В.І. Вступ до програмної інженерії. В 2-х т. Т1. Історія розвитку. Основні поняття: навчальний посібник. Друге видання. – Черкаси: ЧНУ імені Богдана Хмельницького, 2014. – 500 с.
6. Голощапов А. Л. Google Android: программирование для мобильных устройств. – 2-е изд. – СПб.: БХВ-Петербург, 2012. – 448 с.
7. Бочарова Е. История Android: 10 лет с Google. Электронный ресурс (на 12.02.2016): http://rb.ru/story/istoriya-android/
8. История Windows Phone: от Windows Mobile до наших дней. Электронный ресурс (на 12.02.2016): http://zoom.cnews.ru/publication/item/54592
9. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. – СПб.: Питер, 2015. – 1120 с.
10. Назаров С.В., Широков А.И. Современные операционные системы: учебное пособие. – М.: Национальный Открытый Университет «ИНТУИТ», 2012. – 367 с.
11. Столлингс В. Операционные системы. 4-е издание. – М. «Вильямс», 2004. – 848 с.
12. Дейтел Х., Дейтел П., Чофнес Д. Операционные системы: 3-е издание. В 2-х т. Т.1. Основы и принципы. – М.: «Бином-Пресс», 2006, – 1024 с.
13. Деревянко А.С., Солощук М.Н. Операционные системы. Часть 1. Управление ресурсами. Учебное пособие. – Харьков: НТУ «ХПИ», 2002. – 573 с.
14. Дейкстра Э. Взаимодействие последовательных процессов. В кн. «Языки программирования» под ред. Ф.Женюи. – М.: «Мир», 1990. – 506 с.
15. Peterson G.L. Myths about the mutual exclusion problem //Inform. Process. Lett. 12 (3) (1981), pp. 115, 116.
16. Lamport L. A new solution of Dijkstra’s Concurrent Programming Problem // Comm. ACM. 17, 8 (August 1974), pp. 453 – 455.
17. Garg V. K. Concurrent and Distributed Computing in Java. Wiley-IEEE Press, 2004.
18. Авраменко В.С. Реализация примитивов взаимоисключения в ОС. Алгоритм Лэмпорта. Сб. Материалов первой международной научно-технической конференции «Информационные и моделирующие технологии» (Черкассы, 11-12 июня 2008 г.). – Черкассы: ЧНУ им. Б. Хмельницкого, 2008. – С. 5-6.

19. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2001. – 544 с.
20. Дейтел Х., Дейтел П., Чофнес Д. Операционные системы: 3-е изд. В 2-х т. Т.2. Распределенные системы, сети, безопасность. – М.: «Бином-Пресс», 2006. – 704 с.
21. Гордеев А.В. Операционные системы: Учебник для вузов. 2-е изд. – СПб.: Питер, 2007. – 416 с.
22. Карпов В.Е., Коньков К.А. Основы операционных систем. Курс лекций. Учебное пособие. – М.: ИНТУИТ.РУ «Интернет-Университет Информационных Технологий», 2005. – 536 с.
23. Шеховцов В.А. Операційні системи. – К.: Видав. група BHV, 2005. – 576 с.
24. Иртегов Д.В. Введение в операционные системы: Учебное пособие. 2..е изд. – СПб.: БХВ. Петербург, 2008. 1040 с.
25. Востокин С. В. Операционные системы: Учебник. – Самара: Изд-во Самар, гос. аэрокосм, ун-та, 2012. – 120 с.
26. Бондаренко М.Ф., Качко О.Г. Операційні системи: Навчальний посібник. – Х.: Компанія СМІТ, 2008. – 432 с.
27. Безбогов, А.А., Яковлев А.В., Мартемьянов Ю.Ф. Безопасность операционных систем: Учебное пособие. – М.: «Издательство Машиностроение-1», 2007. – 220 с.
28. Коньков К.А. Устройство и функционирование ОС Windows. – М.: Национальный Открытый Университет «ИНТУИТ», 2016. – 240 с.
29. Медник С., Донован Дж. Операционные системы.: Пер. с англ. – М.: Мир, 1978. – 792 с.
30. Хейер Крис и др. Внутренний мир UNIX. – К.: Издательство «Диа-Софт», 1998. – 832 с.
31. Бэкон Дж., Харрис Т. Операционные системы: параллельные и распределенные системы. – СПб.: Питер, 2004. – 799 с.
32. Авраменко В.С., Салапатов В.І. Вступ до програмної інженерії. Том 1. Історія розвитку. Основні поняття. Навчальний посібник. – Черкаси: Черкаський національний університет ім. Б Хмельницького, 2015. – 500 с.
33. Resilient file system (refs) overview. Електронний ресурс (на 02.08.2017): https://docs.microsoft.com/en-us/windows-server/storage/refs/refs-overvie
34. HFS Plus. Интернет-энциклопедия «Википедия». Электронный ресурс. Режим доступа: https://ru.wikipedia.org/wiki/HFS_Plus (на 12.08.2017).
35. B-Дерево. Интернет-энциклопедия «Википедия». Электронный ресурс. Режим доступа: https://ru.wikipedia.org/wiki/B-дерево (на 12.08.2017).
36. Волк С. Mac OS X – UNIX для всех. – М.: Айк. Промоушен, 2002. – 272 с.
37. Пятибратов А.П. Вычислительные системы, сети и телекоммуникации : учебное пособие / А.П. Пятибратов, Л.П. Гудыно, А.А. Кириченко. – М.: КНОРУС, 2013. – 376 с.
38. Воеводин В.В., Воеводин Вл.В.. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 608 с.

Навчальне видання

Авраменко Валентин Семенович
Авраменко Артем Сергійович

ОСНОВИ ОПЕРАЦІЙНИХ СИСТЕМ

Навчальний посібник

Загальний редактор *Г. В. Косенюк*
Комп’ютерне верстання *А. С. Авраменко*

Підписано до друку __.__.2018. Формат 60x84/16.
Ум. друк. арк. 29,00. Тираж 300 пр. Зам. № __

Видавець і виготовлювач
Черкаський національний університет імені Богдана Хмельницького
Адреса: бульвар Шевченка, 81, м. Черкаси, Україна, 18031
Тел. (0472) 37-13-16, факс (0472) 37-22-33,
e-mail: vydav@cdu.edu.ua, http://www.cdu.edu.ua
Свідоцтво про внесення до державного реєстру
суб’єктів видавничої справи ДК №3427 від 17.03.2009 р.