Міністерство освіти і науки України

ДЕРЖАВНИЙ БІОТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

Інститут «Кіберпорт»

Кафедра автоматизації та комп'ютерно-інтегрованих технологій

І. М. Чуб

КОМП'ЮТЕРНІ СИСТЕМИ

Курс лекцій

для здобувачів першого (бакалаврського) рівня вищої освіти денної (заочної) форми навчання за спеціальністю 123 «Комп'ютерна інженерія»

Харків
2025

Міністерство освіти і науки України

ДЕРЖАВНИЙ БІОТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

Інститут «Кіберпорт»

Кафедра автоматизації та комп’ютерно-інтегрованих технологій

І. М. Чуб

КОМП'ЮТЕРНІ СИСТЕМИ

Курс лекцій

для здобувачів першого (бакалаврського) рівня вищої освіти денної (заочної) форми навчання за спеціальністю 123 «Комп’ютерна інженерія»

Затверджено
рішенням науково-методичної ради
інституту «Кіберпорт»
Протокол № 4
від «_23_» __12____ 2024 року

Харків
2025

УДК 510:621.9
Т 41

Схвалено на засіданні кафедри
автоматизації та комп’ютерно-інтегрованих технологій Протокол № 3 від 17.10. 2024 р.

Рецензенти:

С. Я. Бовчалюк, канд. техн. наук, доцент кафедри електронних обчислювальних машин ХНУРЕ.
М. П. Кунденко, д-р. техн. наук, професор, зав. кафедри теплотехніки та енергоефективних технологій НТУ ‘ХПІ’.

Т 41 Комп'ютерні системи : курс лекцій для здобувачів першого (бакалаврського) рівня вищої освіти денної (заочної) форми навчання за спеціальністю 123 «Комп’ютерна інженерія» / І. М. Чуб / – Електрон. дані. – Х. : ДБТУ, 2025. – 120 с.

Конспект лекцій містить 9 тем. Вивчення тем допоможе студентам краще зрозуміти ключові принципи розробки операційних систем та основи побудови інтерфейсів прикладного програмування. Розглядаються базові принципи архітектури операційних систем, а також їх складові та функції, розглядаються питанням управління процесами і потоками з метою синхронізації та організації їхньої взаємодії.

Видання призначене студентам першого (бакалаврського) рівня вищої освіти денної форм навчання спеціальності 123 Комп’ютерна інженерія.

УДК 510:621.9

Відповідальний за випуск: К. В. Демченко, к. т. н., доцент

ЗМІСТ

ВСТУП.....4
1 ОСНОВНІ ПОНЯТТЯ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....5
1.1 Поняття прикладного та системного програмного забезпечення.....5
1.2 Склад системного програмного забезпечення.....5
2 СКЛАД ТА АРХІТЕКТУРА ОПЕРАЦІЙНИХ СИСТЕМ...7
2.1 Склад операційних систем.....7
2.2 Архітектура ОС.....8
3. ПРОЦЕСИ ТА ПОТОКИ.....10
3.1 Концепція процесів і потоків.....10
3.2. Багатозадачність. Форми програмної роботи.....12
3.3 Підсистема управління процесами і потоками.....12
3.4 Роль процесів, потоків і волокон у багатозадачності.....13
3.5 Створення процесів..... 14
3.6 Потоки та їхні моделі..... 16
3.7 Планування та синхронізація процесів і потоків..... 18
4 УПРАВЛІННЯ ПАМ'ЯТТЮ..... 25
4.1 Функції ОС з управління пам'яттю.....25
4.2 Класифікація методів розподілу пам'яті.....26
4.3 Розподіл пам'яті без використання зовнішньої пам'яті..... 27
4.4 Методи структуризації віртуальної пам'яті.....30
5 ФАЙЛОВІ СИСТЕМИ.....34
5.1 Мета та завдання файлової системи.....34
5.2 Організація файлів і доступ до них.....35
5.3 Логічна організація файлу.....36
5.4 Каталожні системи.....37
5.5 Основні можливості файлової системи NTFS..... 38
5.6 Структура тому з файловою системою NTFS.....39
5.7 Можливості NTFS щодо обмеження доступу до файлів і каталогів..... 41
6 УПРАВЛІННЯ ВВЕДЕННЯМ-ВИВЕДЕННЯМ..... 42
6.1 Фізична організація пристроїв введення-виведення..... 42
6.2 Організація програмного забезпечення введення-виведення..... 43
6.3 Обробка переривань..... 44
6.4 Драйвери пристроїв.....44
6.5 Незалежний від пристроїв шар ОС.....44
6.6 Користувацький шар програмного забезпечення.....44
7 ПОБУДОВА ОПЕРАЦІЙНИХ СИСТЕМ.....42
7.1 Принципи побудови операційних систем.....45
7.2 Побудова інтерфейсів операційних систем.....48
7.3 Інтерфейс прикладного програмування.....49

7.4 Класифікація системних викликів.....51
7.5 Інтерфейс користувача..... 53
7.6. Користувацький інтерфейс додатків.....54
7.7 Архітектура, керована подіями.....54
8 UNIX ПОДІБНІ СИСТЕМИ.....55
8.1 Основні поняття системи UNIX.....55
8.1.1 Віртуальна машина.....55
8.2 Операційна система Linux.....58
9 ТРАНСЛЯТОРИ.....59
9.1 Основні принципи побудови трансляторів, компіляторів та інтерпретаторів.....59
9.2 Таблиці ідентифікаторів. Організація таблиць ідентифікаторів.....63
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....120

ВСТУП

У конспекті розглядаються принципи розробки операційних систем та основи побудови інтерфейсів прикладного програмування. Сучасні технології створення системного програмного забезпечення ґрунтуються на розробці програмних додатків, що реалізують користувацький інтерфейс взаємодії з ресурсами обчислювальної системи. Освоєння базових принципів побудови операційних систем і управління їхніми ресурсами дасть змогу розробляти порівняно складні системні програмні додатки.

Особливу увагу в конспекті приділено питанням управління процесами і потоками з метою синхронізації та організації їхньої взаємодії.

Одним із ключових напрямів розробки системного програмного забезпечення є створення програм для керування ресурсами операційної системи. Для цього важливо розуміти основні принципи побудови операційних систем, їхню архітектуру, класифікацію системних ресурсів та методи їх керування.

Також важливим завданням є розробка компонентів інструментальних програмних середовищ, зокрема трансляторів та компіляторів, що використовуються в системах програмування. Особливу увагу приділяють правилам побудови граматик мов програмування.

У цьому конспекті лекцій детально розглядаються базові поняття системного програмного забезпечення, принципи архітектури операційних систем, а також їх складові та функції. Описано управління пам’яттю, створення та керування процесами і потоками, функціонування файлових систем, побудова інтерфейсів сучасних операційних систем та класифікація системних викликів. Також наведено особливості розробки трансляторів і компіляторів, а також управління процесами та потоками.

1 ОСНОВНІ ПОНЯТТЯ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Поняття прикладного та системного програмного забезпечення

Програмне забезпечення (ПЗ) - сукупність програм системи обробки інформації та програмних документів, необхідних для експлуатації цих програм.

Таким чином, ***ПЗ*** - це комплекс програм, що забезпечують обробку або передачу даних і призначені для багаторазового використання та застосування різними користувачами.

Програмне забезпечення є одним із видів забезпечення обчислювальної системи, поряд з апаратним, математичним, інформаційним, лінгвістичним, організаційним і методичним забезпеченням.

Залежно від функціонального призначення можна виділити такі види програмного забезпечення:

- прикладне програмне забезпечення;
- системне програмне забезпечення.

Прикладне програмне забезпечення являє собою комплекс програм для виконання завдань користувача в будь-якій предметній області, орієнтованих на безпосередню взаємодію з користувачем.

Системне програмне забезпечення призначене для виконання двох категорій завдань:

- ефективне управління компонентами обчислювальної системи (процесор, оперативна пам'ять, носії інформації, пристрої введення виведення, мережеве обладнання тощо);
- розробка нового програмного забезпечення.

У деяких випадках, для акцентування ролі засобів, що виконують останню групу завдань, їх виділяють у спеціальну категорію, яка називається інструментальне програмне забезпечення.

1.2 Склад системного програмного забезпечення

В англомовній технічній літературі термін *System Software* (системне програмне забезпечення) означає програми та комплекси програм, які є загальними для всіх, хто спільно використовує технічні засоби комп'ютера, та які застосовуються як для автоматизації розроблення (створення) нових програм, так і для організації виконання програм наявних. З цих позицій системне програмне забезпечення може бути розділене на такі п'ять груп:

- операційні системи;
- системи управління файлами;
- інтерфейсні оболонки для взаємодії користувача з ОС і програмні середовища;
- інструментальні середовища програмування;

- утиліти.

Охарактеризуємо коротко перелічені вище групи.

Операційна система **(ОС)** - комплекс керівних і обробних програм, який, з одного боку, виступає як інтерфейс між апаратурою комп'ютера і користувачем з його завданнями, а з іншого - призначений для найефективнішого використання ресурсів обчислювальної системи та організації надійних обчислень.

Будь-який із компонентів прикладного програмного забезпечення обов'язково працює під управлінням ОС. Операційна система ізолює апаратне забезпечення комп'ютера від прикладних програм користувачів, які взаємодіють зі своїми програмами через інтерфейс ОС. Будь-які команди користувача, перш ніж потрапити в прикладну програму, спочатку проходять через ОС.

Основними функціями, які виконує ОС, є такі:

- приймання від користувача (або від оператора системи) завдань або команд, сформульованих відповідною мовою: у вигляді директив (команд) оператора або у вигляді вказівок (своєрідних команд) за допомогою відповідного маніпулятора (наприклад, за допомогою миші), - та їхнє опрацювання;

- приймання та виконання програмних запитів на запуск, призупинення, зупинення інших програм;

- завантаження в оперативну пам'ять програм, що підлягають виконанню;

- розподіл пам'яті, а також організація віртуальної пам'яті;

- запуск програми (передача їй управління, внаслідок чого процесор виконує програму);

- ідентифікація всіх програм і даних;

- прийом і виконання різних запитів від додатків, що виконуються (операційна система вміє виконувати дуже велику кількість системних функцій або сервісів, які можуть бути запитані з програми, що виконується). Звернення до подібних сервісів здійснюється за низкою правил, які й визначають інтерфейс прикладного програмування (Application Program Interface, API) цієї операційної системи;

- обслуговування всіх операцій введення-виведення;

- забезпечення роботи систем управлінь файлами (СУФ) і/або систем управління базами даних (СУБД), що дає змогу різко збільшити ефективність усього програмного забезпечення;

- забезпечення режиму багатозадачності (мультипрограмування), тобто виконання двох або більше програм на одному процесорі, що створює видимість їх одночасного виконання;

- планування і диспетчеризація завдань відповідно до заданих стратегії та дисциплін обслуговування;

- організація механізмів обміну повідомленнями і даними між програмами, що виконуються;

-Забезпечення взаємодії пов'язаних між собою комп'ютерів (для мережевих ОС);

- захист однієї програми від впливу іншої, забезпечення збереження даних, а також захист самої операційної системи від додатків, що виконуються на комп'ютері;

- автентифікація та авторизація користувачів (автентифікація - процедура перевірки імені користувача та його пароля на відповідність значенням, що зберігаються в його обліковому записі; авторизація - призначення користувачеві, який пройшов автентифікацію, певних прав і привілеїв, що визначають його доступ до тих чи інших системних ресурсів);

- задоволення жорстким обмеженням на час відповіді в режимі реального часу;

- надання послуг на випадок часткового збою системи;

- забезпечення роботи систем програмування, за допомогою яких користувачі готують свої програми.

Призначення *системи управління файлами* - організація більш зручного доступу до даних, організованих як файли. Саме завдяки системі управління файлами замість низькорівневого доступу до даних із зазначенням конкретних фізичних адрес потрібного нам запису використовується логічний доступ із зазначенням імені файлу і запису в ньому.

Як правило, всі сучасні ОС мають відповідні системи управління файлами, а також дозволяють працювати з декількома файловими системами. У цьому разі говорять про монтовані файлові системи (додаткову систему управління файлами можна встановити), і в цьому сенсі вони самостійні.

Слід розуміти, що будь-яка система керування файлами не існує сама по собі - вона розроблена для роботи в конкретній операційній системі та з конкретною файловою системою. Можна сказати, що відома в минулому (і зараз) файлова система FAT (file allocation table) має безліч реалізацій як система керування файлами, наприклад, FAT-16 для самої MS-DOS, FAT32 для Windows 9x і NT тощо. Д (загалом, подібне можна сказати і про NTFS). Інакше кажучи, для роботи з файлами, організованими згідно з деякою файловою системою, для кожної ОС має бути розроблено відповідну систему керування файлами; і ця система керування файлами працюватиме тільки в тій ОС, для якої її і створено.

Для зручності взаємодії з ОС можуть використовуватися **додаткові інтерфейсні оболонки**. Їхнє основне призначення - або розширити можливості з управління ОС, або змінити вбудовані в систему можливості. Як класичні приклади інтерфейсних оболонок і відповідних операційних

середовищ виконання програм можна назвати різні варіанти графічного інтерфейсу X Window у системах сімейства UNIX, різноманітні варіанти інтерфейсів для сімейства ОС Windows компанії Microsoft. Слід зазначити, що в сімействі ОС компанії Microsoft зі спільним інтерфейсом замінною є тільки інтерфейсна оболонка, тоді як саме операційне середовище залишається незмінним - воно інтегроване в ОС. Іншими словами, операційне середовище визначається програмними інтерфейсами, тобто API (application program interface). Інтерфейс прикладного програмування (API) охоплює управління процесами, пам'яттю і введенням/виведенням.

Інструментальне середовище програмування може складатися з таких компонентів:

- система програмування - система, утворена мовою програмування, компіляторами або інтерпретаторами програм, представлених цією мовою, відповідною документацією, а також допоміжними засобами для підготовки програм до форми, придатної для виконання;
- мова програмування високого рівня - формалізована мова для опису розв'язання алгоритму, розв'язання задачі на ПК, поняття і структура якої зручні для сприйняття людиною;
- алгоритмічна мова - штучна мова, призначена для вираження алгоритмів;
- проблемно-орієнтована мова - мова програмування, яка відповідає поняттям певного класу прикладних задач. Проблемно-орієнтована мова зазвичай має набір специфічних образотворчих засобів;
- машинна мова - мова програмування, призначена для подання програм у формі, що дає змогу виконувати її безпосередньо технічними засобами оброблення інформації. Для виконання програми машинною мовою не потрібне застосування трансляторів, компіляторів та інтерпретаторів;
- мова асемблера - мова програмування, яка являє собою символьну форму машинної мови з низкою можливостей, характерних для мов високого рівня.

Будь-яка система програмування може працювати тільки у відповідній ОС, під яку її і створено, однак водночас вона може давати змогу розробляти програмне забезпечення і під інші ОС.

До ***утиліт*** належать спеціальні системні програми, за допомогою яких можна як обслуговувати саму операційну систему, так і готувати для роботи носії даних, виконувати перекодування даних, здійснювати оптимізацію розміщення даних на носії та виконувати деякі інші роботи, пов'язані з обслуговуванням обчислювальної системи. До утиліт слід віднести і програму розбиття жорстких дисків на логічні розділи, і програму форматування, і програму перенесення основних системних файлів самої ОС.

Контрольні питання

1. Які функції виконує ПЗ в обчислювальних системах?
2. Які види програмного забезпечення існують залежно від функціонального призначення?
3. У чому полягає відмінність між прикладним і системним програмним забезпеченням?
4. Які основні завдання виконує системне програмне забезпечення в обчислювальній системі?
5. Для чого призначене інструментальне програмне забезпечення?
6. Які компоненти входять до складу прикладного програмного забезпечення і яка їхня роль?
7. Чому системне програмне забезпечення є важливим для ефективної роботи прикладного програмного забезпечення?

2 СКЛАД ТА АРХІТЕКТУРА ОПЕРАЦІЙНИХ СИСТЕМ

2.1 Склад операційних систем

Функції ОС зазвичай групуються або відповідно до типів локальних ресурсів, якими керує ОС, або відповідно до специфічних завдань, застосовних до всіх ресурсів. Сукупності модулів, що виконують такі групи функцій, утворюють підсистеми операційної системи.

Найважливішими підсистемами управління ресурсами є підсистеми управління процесами, пам'яттю, файлами та зовнішніми пристроями, а підсистемами, загальними для всіх ресурсів, є підсистеми користувацького інтерфейсу, захисту даних та адміністрування.

Управління процесами. Підсистема управління процесами безпосередньо впливає на функціонування обчислювальної системи. Для кожної виконуваної програми ОС організовує один або більше процесів. Кожен такий процес представляється в ОС інформаційною структурою (таблицею, дескриптором, контекстом процесу), яка містить дані про потреби процесу в ресурсах, а також про фактично виділені йому ресурси (області оперативної пам'яті, кількість процесорного часу, файли, пристрої введення-виведення та ін.). Крім того, у цій інформаційній структурі зберігаються дані, що характеризують історію перебування процесу в системі: поточний стан (активний або заблокований), пріоритет, стан регістрів, програмного лічильника тощо.

У сучасних багатозадачних ОС може існувати одночасно кілька процесів, породжених з ініціативи користувачів та їхніх додатків, а також ініційованих ОС для виконання своїх функцій (системні процеси).

Оскільки процеси можуть одночасно претендувати на одні й ті самі

ресурси, підсистема управління процесами планує черговість виконання процесів, забезпечує їх необхідними ресурсами, здійснює взаємодію і синхронізацію процесів.

Керування пам'яттю. Підсистема керування пам'яттю здійснює розподіл фізичної пам'яті між усіма наявними в системі процесами, завантаження і видалення програмних кодів і даних процесів у відведені їм ділянки пам'яті, налаштування адресно-залежних частин кодів процесу на фізичні адреси виділеної ділянки, а також захист ділянок пам'яті кожного процесу.

Одним із найпопулярніших способів керування пам'яттю в сучасних ОС є віртуальна пам'ять. Реалізація механізму віртуальної пам'яті дає змогу програмісту вважати, що в його розпорядженні є однорідна оперативна пам'ять, об'єм якої обмежується тільки можливостями адресації, що надаються системою програмування.

Керування файлами. Функції керування файлами зосереджені у файловій системі ОС. Операційна система представляє окремий набір даних, що зберігаються на зовнішньому накопичувачі, у вигляді файлу - простої неструктурованої послідовності байтів, що мають символьне ім'я. Для зручності роботи з даними файли групуються в каталоги, які, у свою чергу, утворюють групи - каталоги вищого рівня. Файлова система перетворює символьні імена файлів, з якими працює користувач або програміст, у фізичні адреси даних на дисках, організовує спільний доступ до файлів, захищає їх від несанкціонованого доступу.

Керування зовнішніми пристроями. Функції керування зовнішніми пристроями покладаються на підсистему керування зовнішніми пристроями, звану також підсистемою введення-виведення. Вона є інтерфейсом між ядром комп'ютера і всіма підключеними до нього пристроями. Спектр цих пристроїв дуже великий (принтери, сканери, монітори, модеми, маніпулятори, мережеві адаптери, АЦП різного роду та ін.), сотні моделей цих пристроїв різняться набором і послідовністю команд, які використовуються для обміну інформацією з процесором та іншими деталями.

Програма, що керує конкретною моделлю зовнішнього пристрою і враховує всі його особливості, називається драйвером. Наявність великої кількості відповідних драйверів багато в чому визначає успіх ОС на ринку. Створенням драйверів займаються як розробники ОС, так і компанії, що випускають зовнішні пристрої. ОС має підтримувати чітко визначений інтерфейс між драйверами та іншими частинами ОС.

Підтримка високорівневого уніфікованого інтерфейсу прикладного програмування до різноманітних пристроїв введення-виведення є одним із найважливіших завдань ОС. Як правило, такий інтерфейс будується на основі концепції файлового доступу. Її суть полягає в тому, що обмін із

будь-якими зовнішніми пристроями має вигляд обміну з файлами, які мають ім'я і являють собою неструктуровану послідовність байтів.

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

Для кожного користувача системи обов'язкова процедура логічного входу, у процесі якої ОС переконується, що в систему входить користувач, дозволений адміністративною службою. Адміністратор обчислювальної системи визначає й обмежує можливості користувачів у виконанні тих чи інших дій, тобто визначає їхні права щодо звернення та використання ресурсів системи.

Важливим засобом захисту є функції аудиту ОС, що полягає у фіксації всіх подій, від яких залежить безпека системи.

Підтримка відмовостійкості обчислювальної системи реалізується на основі резервування (дискові RAID-масиви, резервні принтери та інші пристрої, інколи резервування центральних процесорів, у ранніх ОС - дуальні та дуплексні системи, системи з мажоритарним органом тощо).

Інтерфейс прикладного програмування. Прикладні програмісти використовують у своїх додатках звернення до операційної системи, коли для виконання тих чи інших дій їм потрібен особливий статус, яким володіє тільки ОС. Можливості операційної системи доступні програмісту у вигляді набору функцій, який називається інтерфейсом прикладного програмування (Application Programming Interface, API). Додатки звертаються до функцій API за допомогою системних викликів. Спосіб, яким додаток отримує послуги операційної системи, дуже схожий на виклик підпрограм.

Спосіб реалізації системних викликів залежить від структурної організації ОС, особливостей апаратної платформи та мови програмування.

Windows API (Application programming interfaces) - загальне найменування цілого набору базових функцій інтерфейсів програмування додатків операційних систем сімейств Windows і Windows NT корпорації "Microsoft". Є найпрямішим способом взаємодії додатків з Windows. Для створення програм, що використовують Windows API, "Microsoft" випускає SDK, який називається Platform SDK і містить документацію, набір бібліотек, утиліт та інших інструментальних засобів (SDK - від англ. Software Development Kit - або "devkit" - комплект засобів розроблення, що дає змогу фахівцям із програмного забезпечення створювати додатки для певного пакета програм, програмного забезпечення базових засобів розроблення, апаратної платформи, комп'ютерної системи, відеоігрових консолей, оперативних систем та інших платформ).

Win32 - 32-розрядний API для сучасних версій Windows. Найпопулярніша нині версія. У сучасних версіях Windows, що походять від

Windows NT, роботу Win32 GUI забезпечують два модулі: csrss.exe (Client/Server Runtime Subsystem), що працює в користувацькому режимі, та win32k.sys у режимі ядра. Роботу ж системних Win32 API забезпечує ядро - ntoskrnl.exe.

Інтерфейс користувача. ОС забезпечує зручний інтерфейс не тільки для прикладних програм, а й для користувача (програміста, адміністратора). Сучасні ОС підтримують розвинені функції користувацького інтерфейсу для інтерактивної роботи за терміналами двох типів: алфавітно-цифровими та графічними.

2.2 Архітектура ОС

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

Більшість сучасних ОС являють собою добре структуровані модульні системи, здатні до розвитку, розширення і перенесення на нові платформи. Якоїсь єдиної уніфікованої архітектури ОС не існує, але відомі універсальні підходи до структурування ОС. Принципово важливими універсальними підходами до розроблення архітектури ОС є:

- модульна організація;
- функціональна надмірність;
- концепція багаторівневої ієрархічної обчислювальної системи, за якою ОС є багатошаровою структурою;
- поділ модулів на дві групи за функціями: ядро - модулі, що виконують основні функції ОС, і модулі, що виконують допоміжні функції ОС;
- поділ модулів ОС на 2 групи за розміщенням у пам'яті обчислювальної системи: резидентні, які постійно перебувають в оперативній пам'яті, і транзитні, які завантажують в оперативну пам'ять тільки на час виконання своїх функцій;
- реалізація двох режимів роботи обчислювальної системи: привілейованого режиму (або режиму ядра - Kernel mode), або режиму супервізора (supervisor mode), і користувацького режиму (user mode), або режиму завдання (track mode);
- обмеження функцій ядра до мінімальної кількості необхідних функцій.

Класичною вважається архітектура операційної системи, заснована на концепції ієрархічної багаторівневої машини, привілейованому ядрі та користувацькому режимі роботи транзитних модулів. Модулі ядра виконують базові функції операційної системи: управління процесами,

пам'яттю, пристроями введення-виведення тощо. Ядро становить серцевину операційної системи, без якої вона є повністю непрацездатною і не може виконати жодну зі своїх функцій. У ядрі розв'язуються внутрішньосистемні задачі організації обчислювального процесу, недоступні для додатка.

Особливий 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 чи інших дій, наприклад, відкриття і читання файлу, отримання системного часу, виведення інформації на дисплей тощо. Функції ядра, які можуть викликатися додатками, утворюють інтерфейс прикладного програмування API.

Для забезпечення високої швидкості роботи ОС модулі ядра здебільшого є резидентними і працюють у привілейованому режимі (Kernel mode). Цей режим забезпечує безпеку роботи самої ОС від втручання додатків, а також можливість роботи модулів ядра з повним набором машинних інструкцій, які дають змогу ядру виконати керування ресурсами комп'ютера (наприклад, перемикання процесора із задачі на задачу, керування пристроями вводу-виводу, розподіл і захист пам'яті тощо).

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

Допоміжні модулі зазвичай поділяють на групи:

- утиліти - програми, що виконують окремі завдання управління і супроводу обчислювальної системи (службові програми та встановлення обладнання);
- системні обробні програми - текстові та графічні редактори тощо;
- програми надання користувачеві додаткових послуг (спеціальні варіанти користувацького інтерфейсу, калькулятор, ігри, засоби мультимедіа тощо);
- бібліотеки процедур різного призначення для спрощення розроблення додатків (наприклад, бібліотека функцій введення-виведення, бібліотека математичних функцій тощо).

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

У концепції багаторівневої (багатошарової) ієрархічної машини структуру ОС також представляють низкою шарів. За такої організації кожен шар обслуговує вищий шар, виконуючи для нього деякий набір функцій, що утворює міжшаровий інтерфейс. На основі цих функцій наступний верхній за ієрархією шар будує свої функції - складніші і

потужніші тощо. Подібна організація системи спрощує її розробку, тому що дає змогу спочатку "зверху вниз" визначити функції шарів і міжшарові інтерфейси, а під час детальної реалізації, рухаючись "від низу до верху" нарощувати потужність функцій шарів. Крім того, модулі кожного шару можна змінювати без необхідності змін в інших шарах (але в жодному разі не змінюючи міжшарових інтерфейсів).

Класична багатошарова архітектура ядра операційної системи може бути представлена схемою, наведеною на рис. 1.

У цій схемі виділено такі шари.

1. Засоби апаратної підтримки ОС. Значна частина функцій ОС може виконуватися апаратними засобами. Суто програмні ОС зараз не існують. Зазвичай у сучасних системах завжди є засоби апаратної підтримки ОС, які прямо беруть участь в організації обчислювальних процесів. До них належать: система переривань, засоби підтримки привілейованого режиму, засоби підтримки віртуальної пам'яті, системний таймер, засіб перемикання контекстів процесів (інформація про стан процесу в момент його зупинення), засоби захисту пам'яті тощо.

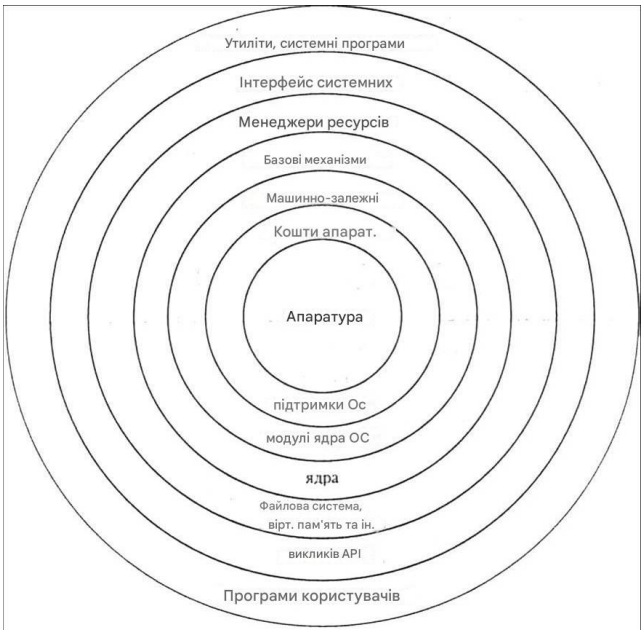

Рис. 1 − Структура ядра операційної системи

2. Машинно-залежні модулі ОС. Цей шар утворює модулі, в яких відображається специфіка апаратної платформи комп'ютера. Призначення цього шару - "екранування" вищерозміщених шарів ОС від особливостей апаратури (у сучасних ОС цей шар називається HAL - Hardware Abstraction Layer - рівень апаратних абстракцій).

3. базові механізми ядра. Цей шар модулів виконує найпримітивніші операції ядра: програмне перемикання контекстів процесів, диспетчер переривань, переміщення сторінок між основною пам'яттю і диском тощо.

Модулі цього шару не ухвалюють рішень про розподіл ресурсів, а тільки обробляють рішення, ухвалені модулями вищих рівнів. Тому їх часто називають виконавчими механізмами для модулів верхніх шарів ОС.

4. Менеджери ресурсів. Модулі цього шару виконують стратегічні завдання з управління ресурсами обчислювальної системи. Це менеджери (диспетчери) процесів, введення-виведення, оперативної пам'яті та файлової системи. Кожен менеджер веде облік вільних і використовуваних ресурсів і планує їх розподіл відповідно до запитів додатків.

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

Підвищення стійкості ОС забезпечується переходом ядра в привілейований режим. При цьому відбувається деяке уповільнення виконання системних викликів. Системний виклик привілейованого ядра ініціює перемикання процесора з призначеного для користувача режиму в привілейований, а при поверненні до додатка - зворотне перемикання. За рахунок цього виникає додаткова затримка в обробці системного виклику. Однак таке рішення стало класичним і використовується в багатьох ОС.

Контрольні питання

1. Які підсистеми операційної системи відповідають за управління ресурсами, і які приклади цих підсистем?
2. Які функції виконує підсистема управління процесами в сучасних багатозадачних ОС?
3. Що таке віртуальна пам'ять і яку роль вона відіграє у керуванні пам'яттю в операційних системах?
4. Як операційна система керує файлами, і які функції виконує файлова система?
5. Яку роль відіграють драйвери у підсистемі керування зовнішніми пристроями?
6. Які дві групи модулів виділяють за функціями в архітектурі операційної системи?

3 ПРОЦЕСИ ТА ПОТОКИ

3.1 Концепція процесів і потоків

Одним з основних понять, пов'язаних з операційними системами, є *процес* - програма, що виконується, включно з поточними значеннями

лічильника команд, регістрів і змінних. З кожним процесом пов'язується його *адресний простір,* що містить саму програму, дані до неї та її стек. Усе програмне забезпечення, що функціонує на комп'ютері, включно з операційною системою, можна уявити набором процесів.

Завданням ОС є управління процесами і ресурсами комп'ютера або, точніше, організація раціонального використання ресурсів в інтересах найефективнішого виконання процесів. Для розв'язання цього завдання операційна система повинна мати інформацію про поточний стан кожного процесу і ресурсу. Універсальний підхід до надання такої інформації полягає у створенні та підтримці таблиць з інформацією щодо кожного об'єкта управління.

Загальне уявлення про це можна отримати з рис. 2, на якому показано чотири різні види таблиць, підтримуваних операційною системою: для пам'яті, пристроїв введення-виведення, файлів (програм і даних) і процесів. Хоча деталі таких таблиць у різних ОС можуть відрізнятися, по суті, всі вони підтримують інформацію за цими чотирма категоріями. Маючи в своєму розпорядженні одні й ті самі апаратні ресурси, але керований різними ОС, комп'ютер може працювати з різним ступенем ефективності. Найбільші складнощі в управлінні ресурсами комп'ютера виникають у мультипрограмних ОС.

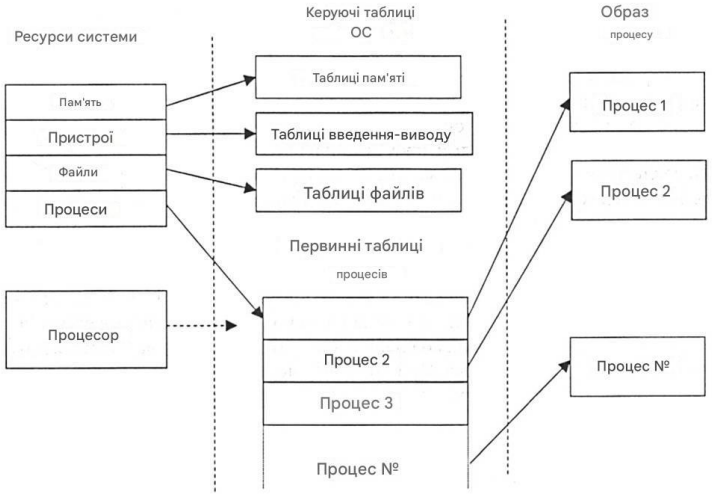

Рис. 2 − Види таблиць, підтримувані операційною системою

Мультипрограмування (багатозадачність, multi-tasking) - це такий спосіб організації обчислювального процесу, за якого на одному процесорі почергово виконуються кілька програм. Щоб підтримувати багатозадачність, ОС має визначити для себе внутрішні одиниці роботи, між якими розділятиметься процесор та інші ресурси комп'ютера. В ОС пакетного опрацювання, поширених у комп'ютерах другого, а спочатку і третього покоління, такою одиницею роботи було завдання. Нині в

більшості операційних систем визначено два типи одиниць роботи: більша одиниця - процес або завдання, і менша - *потік* або *нитка.* Причому процес виконується у формі одного або декількох потоків.

Водночас у деяких сучасних ОС, наприклад у Windows 2000/2003, знову повернулися до такої одиниці роботи, як *завдання* (Job). Завдання в Windows являє собою набір з одного або декількох процесів, керованих як єдине ціле. Зокрема, з кожним завданням асоційовані *квоти* і *ліміти* ресурсів, що зберігаються у відповідному об'єкті завдання. Квоти включають такі пункти, як максимальна кількість процесів (це не дає змоги процесам завдання створювати безконтрольну кількість дочірніх процесів), сумарний час центрального процесора, доступний для кожного процесу окремо та для всіх процесів разом, а також максимальну кількість пам'яті, що використовується, для процесу і всього завдання. Завдання також можуть обмежувати свої процеси в питаннях безпеки, наприклад, забороняти або отримувати права адміністратора навіть за наявності правильного пароля.

Процеси розглядаються операційною системою як заявки або контейнери для всіх видів ресурсів, крім одного - процесорного часу. Цей найважливіший ресурс розподіляється операційною системою між іншими одиницями роботи - потоками, які й дістали свою назву завдяки тому, що вони являють собою послідовності (потоки виконання) команд.

Кожен процес починається з одного потоку, але нові потоки можуть створюватися (породжуватися) процесом динамічно.

Як правило, потік працює в користувацькому режимі, але коли він звертається до системного виклику, то перемикається в режим ядра. Після завершення системного виклику потік продовжує виконуватися в режимі користувача. У кожного потоку є два стеки: один використовується в режимі ядра, інший - у режимі користувача. Крім стану (поточні значення всіх об'єктів потоку), ідентифікатора і двох стеків, кожен потік має контекст (у якому зберігаються його регістри, коли він не працює), приватну область для його локальних змінних, а також може мати власний маркер доступу (інформація про захист). Коли потік завершує свою роботу, він може припинити своє існування. Процес завершується, коли припинить існування останній активний потік.

Перемикання потоків в ОС займає досить багато часу, оскільки для цього необхідно перемикання в режим ядра, а потім повернення в режим користувача. Для надання сильно полегшеного псевдопаралелізму, починаючи з Windows 2000, використовуються волокна (Fiber), подібні до потоків, але заплановані в просторі користувача програмою, яка їх створила. У кожного потоку може бути кілька волокон, з тією різницею, що коли волокно логічно блокується, його поміщають у чергу заблокованих волокон, після чого для роботи вибирають інше волокно в контексті того самого

потоку. При цьому ОС "не знає" про зміну волокон, оскільки той самий потік продовжує роботу.

Таким чином, існує ієрархія робочих одиниць операційної системи, яка стосовно Windows 2000 має такий вигляд (рис. 3).

Виникає запитання: навіщо потрібна така складна організація робіт, яку виконує операційна система? Відповідь потрібно шукати в розвитку теорії та практики мультипрограмування (багатозадачності) з метою максимально ефективного використання головного ресурсу обчислювальної системи - центрального процесора (декількох центральних процесорів)

Рис. 3 − Ієрархія робочих одиниць операційної системи

3.2 Багатозадачність. Форми програмної роботи

Багатозадачність покликана підвищити ефективність використання обчислювальної системи. Однак ефективність може розумітися по-різному. Найхарактернішими показниками ефективності обчислювальних систем є:

- пропускна здатність - кількість завдань, які виконує система за одиницю часу;
- зручність роботи користувачів, що полягає, зокрема, в тому, що вони можуть одночасно працювати в інтерактивному режимі з кількома додатками на одній машині;
- реактивність системи - здатність витримувати заздалегідь задані (можливо, дуже короткі) інтервали часу між запуском програми та отриманням кінцевого результату.

Залежно від вибору показника ефективності ОС поділяють на *системи пакетного опрацювання*, *системи поділу часу* і *системи реального часу*.

Цікава форма мультипрограмної роботи пов'язана з *мультипроцесорною обробкою.* Мультипроцесорне опрацювання - це спосіб організації обчислювального процесу в системі з кількома процесорами, за якого кілька завдань (процесів, потоків) можуть одночасно виконуватися на

різних процесорах системи. Концепція мультипроцесування не нова, вона відома з 70-х років, проте стала доступною в широкому масштабі лише останнього десятиліття, особливо з появою багатопроцесорних ПК.

На відміну від мультипрограмної обробки в мультипроцесорних системах кілька завдань виконують одночасно, тому що є кілька процесорів. Однак це не виключає мультипрограмної обробки на кожному процесорі. При цьому різко ускладнюються всі алгоритми керування ресурсами, а отже - і операційна система.

Мультипроцесорні системи часто характеризують як *симетричні* і як *асиметричні.* Ці терміни належать, з одного боку, до архітектури обчислювальної системи, а з іншого - до способу організації обчислювального процесу.

Симетрична архітектура мультипроцесорної системи передбачає однотипність і одноманітність включення процесорів і велику пам'ять, яка розділяється між цими процесорами. Масштабованість, тобто можливість нарощування кількості процесорів у цьому разі обмежена, тому що всі вони використовують одну й ту саму оперативну пам'ять і, отже, мають розташовуватися в одному корпусі. Така конструкція (масштабування по вертикалі) практично обмежує число процесорів до 4 або 8. У симетричних архітектурах всі процесори користуються однією і тією ж схемою відображення пам'яті, тому можуть швидко обмінюватися даними. Це забезпечує досить високу продуктивність для додатків, у яких кілька завдань повинні активно взаємодіяти між собою (наприклад, під час роботи з базами даних).

У *симетричних* архітектурах обчислювальних систем легко реалізується *симетричне мультипроцесування* спільною для всіх процесорів операційною системою. При цьому всі процесори рівноправно беруть участь і в управлінні обчислювальним процесом, і у виконанні прикладних завдань. Різні процесори можуть у якийсь момент часу одночасно обслуговувати як різні, так і однакові модулі загальної ОС. Для цього програми ОС мають бути *реентерабельними* (повторновхідними).

Операційна система повністю децентралізована. Її модулі виконуються на будь-якому доступному процесорі. Щойно процесор завершує виконання чергового завдання, він передає управління планувальнику завдань. Останній обирає із загальної для всіх процесорів системної черги завдання, яке буде виконуватися на цьому процесорі наступним. Усі ресурси виділяються для кожного виконуваного завдання в міру виникнення потреби в них і не закріплюються за процесором. У вирішенні одного завдання може бути зайнято кілька процесорів, якщо завдання допускає розпаралелювання. У разі відмови одного або більше процесорів симетричні системи порівняно просто реконфігуруються.

В обчислювальних системах з *асиметричною* архітектурою процесори

можуть бути різними як за характеристиками (продуктивність, система команд), так і за функціональною роллю в роботі системи. Наприклад, можуть бути виділені процесори для обчислень, введення-виведення тощо. Ця неоднорідність веде до структурних відмінностей у фрагментах системи, що містять різні процесори (різні схеми підключення, набори периферійних пристроїв, способи взаємодії процесорів із пристроями тощо).

Масштабування в таких системах реалізується інакше, оскільки відсутня вимога єдиного корпусу. Система може складатися з декількох пристроїв, кожен з яких містить один або кілька процесорів. Масштабування в цьому випадку називають горизонтальним, а мультипроцесорну систему - кластерною. У кластерній системі може бути реалізовано тільки асиметричне мультипроцесування з організацією обчислювального процесу за принципом "ведучий - ведений". Цей найпростіший спосіб може бути використаний і в обчислювальних системах із симетричною архітектурою. У таких системах ОС працює на одному процесорі, який називається ведучим і організовує централізоване управління обчислювальним процесом і розподілом усіх ресурсів системи.

3.3 Підсистема управління процесами і потоками

Однією з основних підсистем будь-якої сучасної мультипрограмної ОС, що безпосередньо впливає на функціонування комп'ютера, є підсистема управління процесами і потоками. Основними функціями цієї підсистеми є:

- створення процесів і потоків;
- забезпечення процесів і потоків необхідними ресурсами;
- ізоляція процесів;
- планування виконання процесів і потоків (узагалі слід говорити і про планування завдань);
- диспетчеризація потоків;
- організація міжпроцесної взаємодії;
- синхронізація процесів і потоків;
- завершення і знищення процесів і потоків.

До створення процесу призводять п'ять основних подій:

1) ініціалізація ОС (завантаження);

2) виконання запиту працюючого процесу на створення процесу;

3) запит користувача на створення процесу, наприклад під час входу в систему в інтерактивному режимі;

4) ініціювання пакетного завдання;

5) створення операційною системою процесу, необхідного для роботи будь-яких служб.

З технічного погляду в усіх перерахованих випадках новий процес формується однаково: поточний процес виконує системний запит на створення нового процесу.

Для того щоб процеси не могли втрутитися в розподіл ресурсів, а також не могли пошкодити коди і дані один одного, найважливішим завданням ОС є ізоляція одного процесу від іншого. Для цього операційна система забезпечує кожен процес окремим віртуальним адресним простором, так що жоден процес не може отримати прямого доступу до команд і даних іншого процесу.

Перехід від виконання одного потоку до іншого здійснюється в результаті *планування і диспетчеризації.* Робота з визначення того, в який момент необхідно перервати виконання поточного потоку і якому потоку надати можливість виконатися, називається плануванням. Під час планування береться до уваги пріоритет потоків, час їхнього очікування в черзі, накопичене виконання, інтенсивність звернення до введення-виведення та інші чинники.

Диспетчеризація полягає в реалізації знайденого в результаті планування рішення, тобто в перемиканні процесора з одного потоку на інший. Диспетчеризація проходить у три етапи:

- збереження контексту поточного потоку;
- завантаження контексту потоку, обраного в результаті планування;
- запуск нового потоку на виконання.

Синхронізація потоків є однією з найважливіших функцій підсистеми керування процесами та потоками. Сучасні операційні системи надають безліч механізмів синхронізації, зокрема семафори, м'ютекси, критичні області та події. Усі ці механізми працюють із потоками, а не процесами. Тому коли потік блокується на семафорі, інші потоки цього процесу можуть продовжувати роботу.

Щоразу, коли процес завершується, а це відбувається завдяки одній із таких подій: звичайний вихід, вихід за помилкою, вихід за невиправною помилкою, знищення іншим процесом, - ОС робить кроки, щоб "зачистити сліди" його перебування в системі. Підсистема управління процесами закриває всі файли, з якими працював процес, звільняє області оперативної пам'яті, відведені під дані і системні інформаційні структури процесу. Виконується корекція всіляких черг ОС і списку ресурсів, у яких були посилання на процес, що завершується.

3.4 Роль процесів, потоків і волокон у багатозадачності

У найпростішому випадку процес складається з одного потоку, і в деяких сучасних ОС зберігається такий стан. Мультипрограмування в таких ОС здійснюється на рівні процесів. За необхідності взаємодії процеси звертаються до операційної системи, яка, виконуючи функції посередника, надає їм засоби міжпроцесного зв'язку - канали, поштові акції, секції пам'яті, які можна розділити, тощо.

Але сучасній операційній системі поряд із процесами потрібен інший

механізм розпаралелювання обчислень, який враховував би тісні зв'язки між окремими гілками обчислень однієї і тієї ж програми. Для цих цілей сучасні ОС пропонують механізм багатопотокової обробки (multithreading).

Поняттю "потік" відповідає послідовний перехід процесора від однієї команди до іншої. Процесу ОС призначають адресний простір і набір ресурсів, які спільно використовуються всіма його потоками. На відміну від процесів, які належать, взагалі кажучи, конкуруючим додаткам, усі потоки одного процесу завжди належать одному додатку, тому ОС ізолює потоки значно меншою мірою, ніж процеси в традиційній мультипрограмній системі. Усі потоки одного процесу використовують спільні файли, таймери, пристрої, одну й ту саму частину оперативної пам'яті, один і той самий адресний простір. Це означає, що вони поділяють одні й ті самі глобальні змінні. Оскільки кожен потік може мати доступ до будь-якої віртуальної адреси, один потік може використовувати стек іншого потоку. Між потоками одного процесу немає повного захисту, по-перше, тому що це неможливо, а по-друге, не потрібно. Щоб організувати взаємодію та обмін даними, потокам не потрібно звертатися до ОС, їм достатньо використовувати загальну пам'ять - один потік записує дані, а інший читає їх. З іншого боку, потоки різних процесів, як і раніше, добре захищені один від одного.

Таким чином, мультипрограмування ефективніше на рівні потоків, а не процесів. Ще більший ефект багатопотокової обробки досягається в мультипроцесорних системах, у яких потоки можуть виконуватися на різних процесорах дійсно паралельно.

3.5 Створення процесів

Створити процес - це насамперед створити опис процесу: кілька інформаційних структур, що містять усі відомості (атрибути) про процес, необхідні операційній системі для управління ним. До числа таких відомостей можуть входити: ідентифікатор процесу, дані про розташування в пам'яті виконуваного модуля, ступінь привілейованості процесу (пріоритет і права доступу) тощо.

Прикладами таких описувачів процесу є:

- керуючий блок процесу (PCB -Process Control Block) в OS/2;
- дескриптор процесу в UNIX;
- об'єкт-процес (object-process) у Windows NT/2000/2003.

Створення процесу включає завантаження кодів і даних виконуваної програми цього процесу з диска в операційну пам'ять. Для цього потрібно знайти цю програму на диску, перерозподілити оперативну пам'ять і виділити пам'ять виконуваній програмі нового процесу. Крім того, під час роботи програми зазвичай використовується стек, за допомогою якого реалізуються виклики процедур і передача параметрів.

Множина, до якої входять програма, дані, стеки та атрибути процесу, називається образом процесу.

Типові елементи образу процесу наведено в табл. 1.

Таблиця 1− Елементи образу процесу

Інформація	Опис
Дані користувача	Змінна частина користувацького адресного простору (дані програми, користувацький стек, код, що модифікується)
Користувацька програма	Програма, яку необхідно виконати
Системний стек	Один або кілька системних стеків для зберігання параметрів і адрес виклику процедур і системних служб
Керуючий блок процесу	Дані, необхідні операційній системі для управління процесом

Місцезнаходження образу процесу залежить від використовуваної схеми керування пам'яттю. У більшості сучасних ОС з віртуальною пам'яттю образ процесу складається з набору блоків (сегменти, сторінки або їх комбінація), необов'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 міститься повний образ. Коли в основну пам'ять завантажується частина образу, вона туди не переноситься, а копіюється. Однак якщо частина образу в основній пам'яті модифікується, вона має бути скопійована на диск.

Під час керування процесами ОС використовує два основні типи інформаційних структур: *блок керування процесом (дескриптор* процесу) і *контекст* процесу. Дескриптори процесів об'єднуються в таблицю процесів, яка розміщується в області ядра. На підставі інформації, що міститься в таблиці процесів, ОС здійснює планування і синхронізацію процесів.

У дескрипторі (блоці управління) процесу міститься така інформація про процес, яка необхідна ядру протягом усього життєвого циклу процесу незалежно від того, перебуває він в активному або пасивному стані, і знаходиться образ в оперативній пам'яті або на диску. Цю інформацію можна розділити на три категорії:

- інформація щодо ідентифікації процесу;
- інформація щодо стану процесу;
- інформація, яка використовується під час управління процесом.

Кожному процесу присвоюється числовий ідентифікатор, який може бути просто індексом у первинній таблиці процесів. У будь-якому разі має бути деяке відображення, що дає змогу операційній системі знайти за

ідентифікатором процесу відповідні йому таблиці. Під час створення нового процесу ідентифікатори вказують *батьківський* і *дочірні* процеси. Крім того, процесу може бути присвоєно ідентифікатор користувача, який вказує, хто з користувачів відповідає за це завдання.

Інформація щодо стану та управління процесом включає такі основні дані:

- стан процесу, що визначає готовність процесу до виконання (що виконується, готовий до виконання, що очікує якоїсь події, призупинений);
- дані про пріоритет (поточний пріоритет, за замовчуванням, максимально можливий);
- інформація про події - ідентифікація події, настання якої дасть змогу продовжити виконання процесу;
- покажчики, що дають змогу визначити розташування образу процесу в оперативній пам'яті та на диску;
- покажчики на інші процеси (зокрема ті, що перебувають у черзі на виконання);
- прапори, сигнали та повідомлення, що стосуються обміну інформацією між двома незалежними процесами;
- дані про привілеї, що визначають права доступу до певної області пам'яті або можливості виконувати певні види команд, використовувати системні утиліти та служби;
- покажчики на ресурси, якими керує процес (наприклад, перелік відкритих файлів);
- відомості з історії використання ресурсів і процесора;
- інформація, пов'язана з плануванням. Ця інформація багато в чому залежить від алгоритму планування. Сюди належать, наприклад, такі дані, як час очікування або час, протягом якого процес виконувався під час останнього запуску, кількість виконаних операцій вводу-виводу тощо.

Контекст процесу містить інформацію, що дає змогу системі призупиняти і відновлювати виконання процесу з перерваного місця.

У контексті процесу міститься така основна інформація:

- вміст регістрів процесора (зазвичай 8-32 регістри і до 100 у RISC-процесорах) доступних користувачеві;
- вміст лічильника команд;
- стан керуючих регістрів і регістрів стану;
- коди умов, що відображають результат виконання останньої арифметичної або логічної операції (наприклад, знак рівності нулю, переповнення);
- покажчики вершин стеків, що зберігають параметри та адреси виклику процедур і системних служб.

Слід зауважити, що частина цієї інформації, відома як "слово стану програми" (Program Status Word - PSW), фіксується в спеціальному регістрі

процесора (наприклад, у регістрі EFLAGS у процесорах Pentium).

Найпростішу модель процесу можна побудувати виходячи з того, що в будь-який момент часу процес або виконується, або не виконується, тобто має тільки два стани (рис. 4).

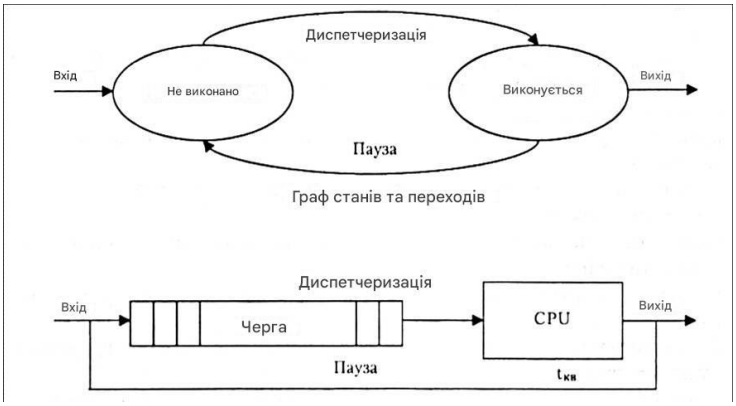

Рис. 4 − Модель процесу для двох станів

Якби всі процеси були завжди готові до виконання, то черга за цією схемою могла б працювати цілком ефективно. Така черга працює за принципом обробки в порядку надходження, а процесор обслуговує наявні процеси круговим методом (Round-robin). Кожному процесу відводиться певний проміжок часу, після закінчення якого він повертається в чергу.

Однак у такому простому прикладі подібна реалізація не є адекватною: частина процесів готова до виконання, а частина заблокована, наприклад, через очікування введення-виведення. Тому за наявності однієї черги диспетчер не може просто вибрати для виконання перший процес із черги. Перед цим він повинен буде переглянути весь список, відшукуючи незаблокований процес, що перебуває в черзі найдалі від інших.

Тому видається природним розділити всі процеси, що не виконуються, на два типи: готові до виконання і заблоковані. Корисно додати ще два стани, як показано на рис. 5.

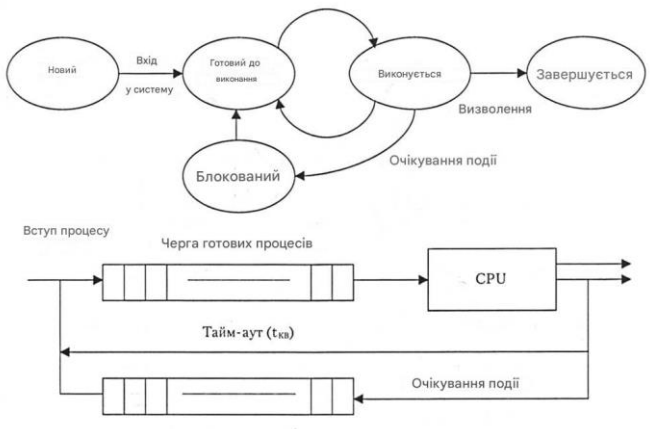

Рис. 5 − Модель процесу для чотирьох станів

У чому переваги і недоліки такої моделі, і як усунути ці недоліки? Оскільки процесор працює набагато швидше за виконання операцій введення-виведення, то незабаром усі процеси, що перебувають у пам'яті, опиняються в стані очікування введення-виведення. Таким чином, процесор може простоювати навіть у багатозадачній системі. Що робити? Можна збільшити ємність основної пам'яті, щоб у ній уміщалося більше процесів.

Інше розв'язання проблеми - свопінг - перенесення частини процесів з оперативної пам'яті на диск і завантаження іншого процесу з черги призупинених (але не заблокованих!) процесів, які перебувають у зовнішній пам'яті.

3.6 Потоки та їхні моделі

При створенні потоків, так само як і при створенні процесів, ОС генерує спеціальну інформаційну структуру - *описувач потоку*, який містить ідентифікатор потоку, дані про права доступу та пріоритет, про стан потоку та іншу інформацію.

Описувач потоку можна розділити на дві частини: *атрибути* блоку управління і *контекст* потоку. Зауважимо, що в разі багатопотокової системи процеси контексту не мають, тому що їм не виділяється процесор.

Є два способи реалізації пакета потоків:

- у просторі користувача або на рівні користувача (user-level threads - ULT);
- у ядрі або на рівні ядра (kernel-level threads - KLT).

У програмі, що повністю складається з ULT-потоків, усі дії з управління потоками виконуються самим додатком. Ядро про потоки нічого не знає і керує звичайними однопотоковими процесами (рис. 6).

Найбільш явна перевага цього підходу полягає в тому, що пакет потоків на рівні користувача можна реалізувати навіть в ОС, яка не підтримує потоки.

Якщо керування потоками відбувається в просторі користувача, кожному процесу необхідна власна таблиця потоків. Вона аналогічна таблиці процесів, з лише різницею, що відстежує такі характеристики потоків, як лічильник команд, покажчик вершини стека, регістри стану тощо. Коли потік переходить у стан готовності або блокування, вся інформація, необхідна для повторного запуску, зберігається в таблиці потоків.

За замовчуванням, додаток на початку своєї роботи складається з одного потоку і його виконання починається як виконання цього потоку. Такий застосунок разом зі складовим його потоком розміщується в одному процесі, який керується ядром.

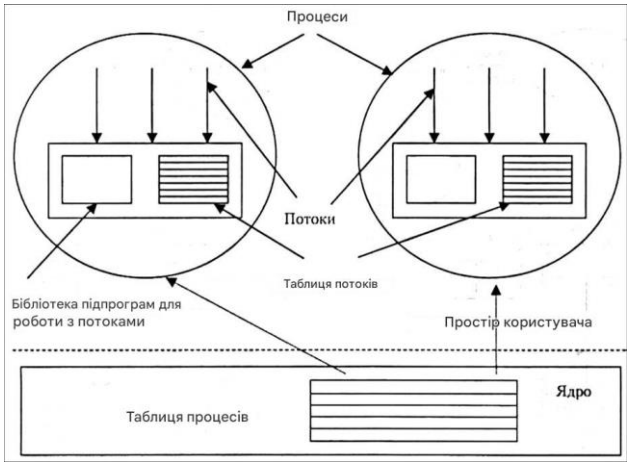

Рис. 6 − Реалізація пакета потоків у просторі користувача

Потік, що виконується, може породити новий потік, який виконуватиметься в межах того самого процесу. Новий потік створюється за допомогою виклику спеціальної підпрограми з бібліотеки, призначеної для роботи з потоками. Передача керування цій програмі відбувається внаслідок виклику відповідної процедури.

Бібліотека підпрограм для роботи з потоками створює структуру даних для нового потоку, а потім передає керування одному з готових до виконання потоків цього процесу, керуючись деяким алгоритмом планування.

Усі ці події відбуваються в користувацькому просторі в рамках одного процесу. Ядро навіть "не підозрює" про цю діяльність і продовжує здійснювати планування процесу як єдиного цілого та приписувати йому єдиний стан виконання.

Використання потоків на рівні користувача має такі переваги:

1) висока продуктивність, оскільки для керування потоками процесу не потрібно перемикатися в режим ядра і назад. Процедура, що зберігає інформацію про потік, і планувальники є локальними процедурами, їхній виклик істотно ефективніший, ніж виклик ядра;

2) є можливість використання різних алгоритмів планування потоків у різних додатках (процесах) з урахуванням їхньої специфіки;

3) використання потоків на користувацькому рівні можна застосувати для будь-якої операційної системи. Для їхньої підтримки в ядро системи не потрібно вносити будь-які зміни.

Однак є й недоліки порівняно з використанням потоків на рівні ядра:

1) у типовій ОС багато системних викликів є блокувальними. Коли в потоці, що тане на користувацькому рівні, виконується системний виклик, блокується не тільки цей потік, а й усі потоки того процесу, до якого він належить;

2) у стратегії з наявністю потоків тільки на користувацькому рівні додаток не може скористатися перевагою багатопроцесорної системи, тому що ядро закріплює за кожним процесом тільки один процесор. Тому кілька потоків одного і того ж процесу не можуть виконуватися одночасно;

3) під час запуску одного потоку жоден інший потік не буде запущено, поки перший добровільно не віддасть процесор. Усередині одного процесу немає переривань за таймером, унаслідок чого неможливо створити планувальник для почергового виконання потоків.

Розглянемо тепер потоки на рівні ядра. У цьому випадку в області додатка система підтримки виконання програм не потрібна, немає необхідності і в таблицях потоків у кожному процесі. Замість цього є єдина таблиця потоків, що відстежує всі потоки в системі. Якщо потоку необхідно створити новий потік або завершити наявний, він виконує запит ядра, який створює або завершує потік, вносячи зміни в таблицю потоків (рис. 7).

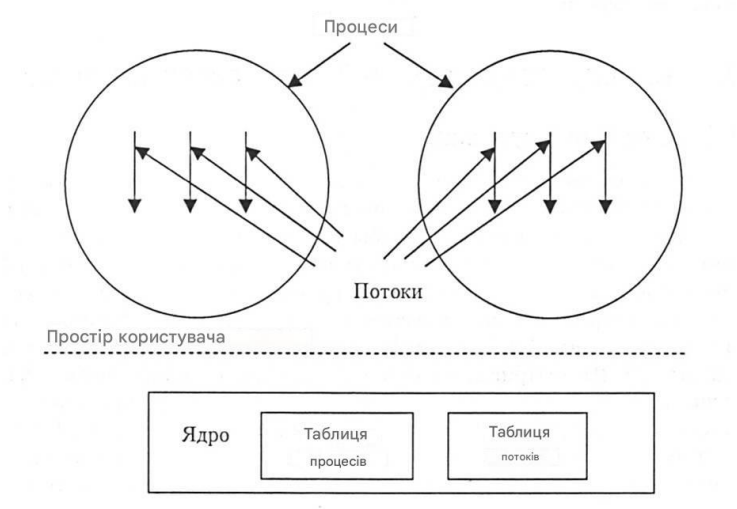

Рис. 7 − Реалізація пакета потоків на рівні ядра

Будь-який додаток можна запрограмувати як багатопоточний, при цьому всі потоки додатка підтримуються в рамках єдиного процесу. Ядро підтримує інформацію контексту процесу як єдиного цілого, а також контексти кожного окремого потоку процесу. Планування здійснюється ядром виходячи зі стану потоків.

За допомогою такого підходу вдається позбутися основних недоліків потоків користувацького рівня:

1. Можливе планування роботи декількох потоків одного і того ж процесу на декількох процесорах.

2. Реалізується мультипрограмування в режимах декількох процесів.

3. У разі блокування одного з потоків процесу ядро може вибрати для виконання інший потік цього ж процесу.

4. Процедури ядра можуть бути багатопотоковими.

Головний недолік пов'язаний із необхідністю дворазового перемикання режимів "користувацький - ядро, ядро - користувацький" для передання одного потоку до іншого в межах одного й того самого процесу.

З метою поєднання переваг реалізації потоків на рівні користувача і на рівні ядра було виправдано багато способів змішаної реалізації. Один із методів полягає у використанні керування ядром і подальшому мультиплексуванні потоків на рівні користувача.

У такій моделі ядро знає тільки про потоки свого рівня і керує ними. Деякі з цих потоків можуть містити по кілька потоків користувацького рівня, мультиплексованих поверх них. Потоки користувацького рівня створюються, виконуються, завершуються і управляються так само, як потоки рівня користувача в процесі, запущеному в системі, що не підтримує багатопоточність. Передбачається, що у кожного потоку ядра є набір потоків на рівні користувача, які використовують його по черзі.

3.7 Планування та синхронізація процесів і потоків

3.7.1 Види планування

Основна мета планування обчислювального процесу полягає в розподілі часу процесора (декількох процесорів) між завданнями користувачів, які виконуються, у такий спосіб, щоб задовольняти вимоги, які висувають користувачі до обчислювальної системи. Такими вимогами можуть бути, як це вже зазначалося, пропускна здатність, час відгуку, завантаження процесора тощо.

Усі види планування, що використовуються в сучасних ОС, залежно від часового масштабу поділяються на *довгострокове, середньострокове, короткострокове* і планування введення-виведення (табл. 2).

Таблиця 2 − Елементи образу процесу

Вид планування	Функції, що виконуються
Довгострокове	Рішення про додавання завдання (процесу) в пул виконуваних у системі
Середньострокове	Рішення про додавання процесу до числа процесів, повністю або частково розміщених в основній пам'яті
Короткострокове	Рішення про те, який із доступних процесів (потоків) буде виконуватися процесором
Планування введення-виведення	Рішення про те, який із запитів процесів (потоків) на операцію введення-виведення буде виконуватися вільним пристроєм введення-виведення

Короткостроковий планувальник, який часто називають *диспетчером*

(dispatcher), працює найчастіше, визначаючи, який процес або потік буде виконуватися наступним. Нижче наведено перелік функцій, які виконує планувальник кожного виду.

У більшості операційних систем універсального призначення планування здійснюється *динамічно* (on-line), тобто рішення приймаються під час роботи системи на основі аналізу поточної ситуації. Знайдене в таких умовах рішення рідко є оптимальним.

Інший тип планування - *статичний* (попередній) може бути використаний тільки в спеціалізованих системах із заданим набором завдань (заздалегідь визначеним), наприклад, у керуючих обчислювальних системах або системах реального часу. У цьому разі статичний планувальник (або попередній планувальник) ухвалює рішення не під час роботи системи, а заздалегідь (off-line).

Результатом його роботи є розклад - таблиця, у якій зазначено, якому процесу, коли і на який час має бути надано процесор. При цьому накладні витрати ОС на виконання розкладу значно менші, ніж при динамічному плануванні.

Короткостроковий планувальник (диспетчер) реалізує знайдене рішення, тобто перемикає процесор з одного процесу (потоку) на інший. Він викликається в разі настання події, яка може призупинити поточний процес або надати можливість припинити виконання цього процесу (потоку) на користь іншого. Прикладами цих подій можуть бути:

- переривання таймера;
- переривання введення-виведення;
- виклики операційної системи;
- сигнали.

Диспетчеризація зводиться до такого:

- збереження контексту поточного потоку, який потрібно змінити;
- завантаження контексту нового потоку, обраного в результаті планування;
- запуск нового потоку на виконання.

3.7.2 Алгоритми планування потоків

У багатозадачній системі потік (процес, якщо операційна система працює тільки з процесами) може перебувати в одному з трьох основних станів:

- виконання - активний стан потоку, під час якого потік володіє всіма необхідними ресурсами і безпосередньо виконується процесором;
- очікування - пасивний стан потоку, перебуваючи в якому потік заблокований через свої внутрішні причини (чекає на здійснення певної події, наприклад, завершення операції введення-виведення, отримання повідомлення від іншого потоку або звільнення будь-якого необхідного

йому ресурсу);

- готовність - також пасивний стан потоку, але в цьому випадку потік заблокований у зв'язку із зовнішньою щодо нього обставиною (має всі необхідні ресурси, готовий виконуватися, але процесор зайнятий виконанням іншого потоку).

Протягом свого життя кожен потік переходить з одного стану в інший відповідно до алгоритму планування потоків, прийнятого в даній операційній системі. Типовий граф станів потоку має вигляд, наведений на рис. 8.

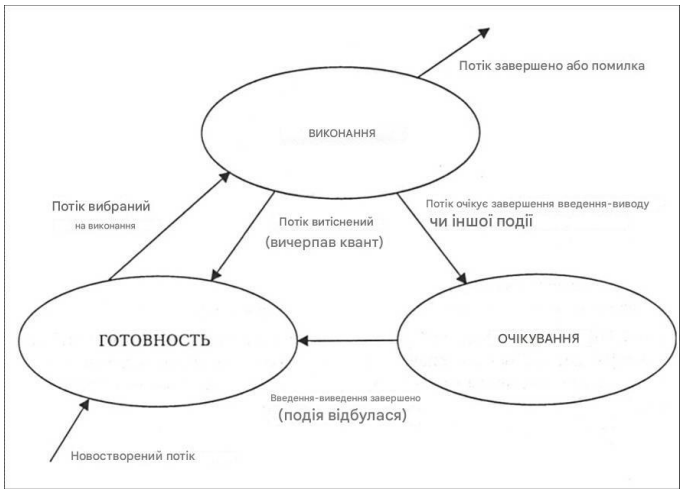

Рис. 8 − Граф станів потоку

Із найзагальніших позицій усю множину алгоритмів планування можна поділити на два класи: алгоритми планування, що витісняють і невитісняють, і алгоритми планування, що не витісняють.

Невитісняючі (non-preemptive) алгоритми засновані на тому, що активному потоку дозволяється виконуватися, доки він сам, за своєю ініціативою, не віддасть управління операційній системі для того, щоб вона вибрала з черги готовий до виконання потік.

Витісняючі (preemptive) алгоритми - це такі способи планування потоків, у яких рішення про перемикання процесора з виконання одного потоку на виконання іншого потоку ухвалюють за операційною системою, а не за активним завданням.

У першому випадку механізм планування розподіляється між операційною системою і прикладними програмами. У другому випадку функції планування потоків цілком зосереджені в операційній системі.

Недоліком першого типу алгоритмів планування є необхідність розроблення такого додатка, який буде "дружнім" щодо інших виконуваних одночасно з ним програм. Тому розподіл функцій планування між ОС і додатками досить складний у програмістському відношенні і використовується, як правило, у спеціалізованих системах з фіксованим набором завдань. Водночас істотною перевагою невитісняючого планування

є вища швидкість перемикання потоків.

Хорошим прикладом вдалої реалізації невитісняючого планування потоків може слугувати ОС NetWare 3.x і 4.x, у яких завдяки такому плануванню досягнуто високої швидкості виконання файлових операцій.

Однак майже у всіх сучасних ОС (UNIX, Windows NT/2000/2003, OS/2, VAX/VMS та ін.) реалізовано алгоритми планування, що витісняють. В основі багатьох таких алгоритмів лежить концепція квантування. Відповідно до неї кожному потоку по черзі для виконання надається обмежений безперервний період процесорного часу - квант.

Зміна активного потоку відбувається, якщо:

- потік завершується і покинув систему;
- сталася помилка;
- потік перейшов у стан очікування;
- вичерпано квант часу, відведений цьому потоку.

Потік, який вичерпав свій квант, переводиться в стан готовності й очікує, коли йому буде надано новий квант процесорного часу, а на виконання згідно з визначеним правилом обирається новий потік із черги готових потоків (рис. 9).

Особливості алгоритмів, заснованих на квантуванні.

1) Перемикання контекстів потоків пов'язане з втратами процесорного часу**,** які не залежать від величини кванта, але залежать від частоти перемикання, яким чим більший квант, тим менші сумарні витрати процесорного часу на перемикання потоків.

2) Зі збільшенням кванта може бути погіршена якість обслуговування користувачів, пов'язана зі зростанням часу реакції системи.

3) В алгоритмах, заснованих на квантуванні, ОС не має жодних відомостей про розв'язувані задачі (довгі чи короткі, інтенсивно введення-виведення чи ні, важливе швидке виконання чи ні і т. п.). Диференціація обслуговування при квантуванні базується на "історії існування" потоку в системі.

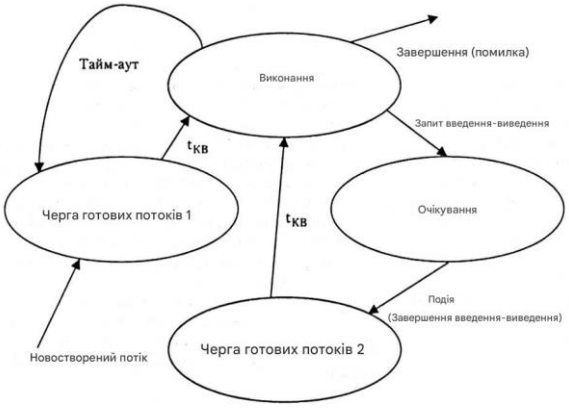

Рис. 9 − Граф станів потоку під час квантування

3.7.3 Алгоритми пріоритетного планування

Важливою концепцією, що лежить в основі багатьох витісняючих алгоритмів планування, є пріоритетне обслуговування. Воно передбачає наявність у потоків деякої спочатку відомої характеристики - пріоритету, на підставі якого визначається порядок виконання потоків. Що вищий пріоритет, то вищі привілеї потоку, то менше часу потік перебуває в чергах. Пріоритет задається числом (цілим або дробовим, позитивним або негативним).

У більшості операційних систем, що підтримують потоки, пріоритет потоку пов'язаний із пріоритетом процесу, у межах якого виконується потік. Пріоритет процесу призначається операційною системою при його створенні, його значення включається в дескриптор процесу і використовується при призначенні пріоритету потоку цього процесу. Під час призначення пріоритетів новоствореному процесу ОС враховується, чи є цей процес системним або прикладним, яким є статус користувача, що запустив його (адміністратор, користувач, частина тощо), чи була явна вказівка користувача про присвоєння процесу певного рівня пріоритету.

Потік може бути ініційований не тільки за командою користувача, а й у результаті виконання системного виклику іншим потоком. У цьому випадку ОС враховує значення параметрів системного виклику.

Зміни пріоритету можуть відбуватися з ініціативи самого потоку, коли він звертається з відповідним викликом до ОС, або з ініціативи користувача, коли він виконує відповідну команду. Крім цього, сама ОС може змінити пріоритети потоків залежно від ситуації, що складається в системі. В останньому випадку пріоритети називаються динамічними на відміну від незмінних, фіксованих пріоритетів. Можливості користувачів впливати на пріоритети процесів і потоків обмежені ОС. Зазвичай це дозволяється адміністраторам, і то в певних межах. У більшості випадків ОС присвоює пріоритети потокам за замовчуванням.

Існують два різновиди пріоритетного планування: з відносними та абсолютними пріоритетами. В обох випадках на виконання вибирають потік, що має найвищий пріоритет. Але визначення моменту зміни активного потоку вирішується по-різному. У системах з відносними пріоритетами активний потік виконується доти, доки він сам не залишить процесор, перейшовши в стан очікування (по введенню-виведенню, наприклад), або не завершиться, або станеться помилка. У системах з абсолютними пріоритетами виконання активного потоку переривається ще й через появу потоку, що має вищий пріоритет, ніж у активного потоку. У цьому разі перерваний потік переходить у стан готовності.

Як приклад розглянемо організацію пріоритетного обслуговування в Windows 2000/2003 (W2K). Тут пріоритети організовані у вигляді двох груп,

або класів: реального часу і змінні. Кожна з груп складається з 16 рівнів пріоритетів (рис. 10). Потоки, що потребують негайної уваги, перебувають у класі реального часу, який включає такі функції, як здійснення комунікацій і завдання реального часу.

Загалом, оскільки W2K використовує планувальник витіснення з урахуванням пріоритетів, потоки з пріоритетами реального часу мають перевагу щодо інших потоків. В однопроцесорній системі, коли стає готовим до виконання потік із вищим пріоритетом, ніж той, що виконується зараз, поточний потік витісняється і починає виконуватися більш високопріоритетний потік.

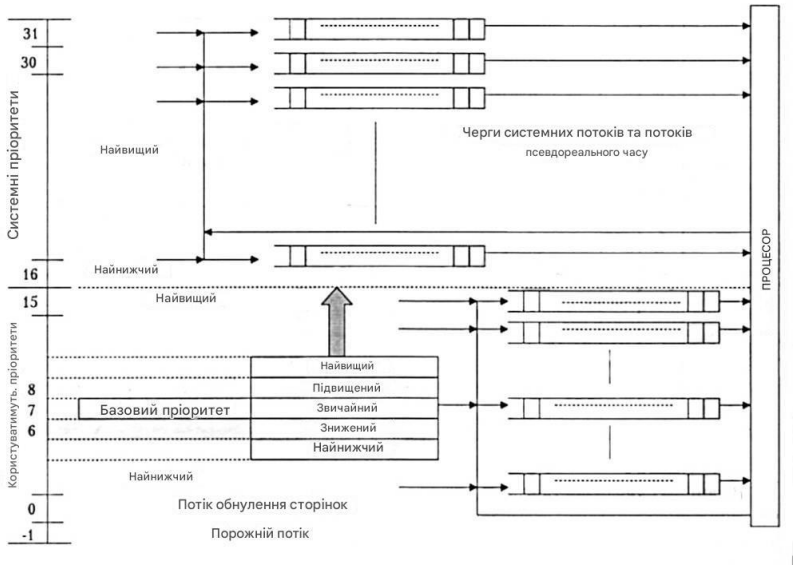

Рис. 10 − Організація пріоритетного обслуговування в Windows 2000

Пріоритети з різних класів обробляються трохи по-різному. У класі пріоритетів реального часу всі потоки мають фіксований пріоритет (від 16 до 31), який ніколи не змінюється, і всі активні потоки з певним рівнем пріоритету розташовуються в круговій черзі цього класу (t_{KB}= 20 мс для W2K Professional, 120 мс - для однопроцесорних серверів).

У класі змінних пріоритетів потік починає роботу з базового пріоритету процесу, який може набувати значення від 1 до 15. Кожен потік, пов'язаний із процесом, має свій базовий пріоритет, який дорівнює базовому пріоритету процесу, або відрізняється від нього не більше ніж на 2 рівні в більший або менший бік. Після активації потоку його динамічний пріоритет може коливатися в певних межах.

Коли ж збільшується пріоритет потоку?

1. Коли завершується операція введення-виведення і звільняється потік, що очікує її, його пріоритет збільшується, щоб дати шанс цьому потоку запуститися швидше і знову запустити операцію введення-виведення. Суть у тому, щоб підтримати зайнятість пристроїв введення-виведення.

Величина, на яку збільшується пріоритет, залежить від пристрою введення-виведення. Як правило, це 1 - для диска, 2 - для послідовної лінії, 6 - для клавіатури і 8 - для звукової карти.

2. Якщо потік чекав на семафор, м'ютекс або іншу подію, то коли він відпускається, до його пріоритету додаються 2 одиниці, якщо це потік переднього плану (тобто управляє вікном, до якого спрямовується введення з клавіатури), та 1 одиниця в іншому разі. Таким чином, інтерактивний процес отримує перевагу перед іншими процесами. Нарешті, якщо потік графічного інтерфейсу користувача прокидається, тому що стало доступним віконне введення, він також отримує надбавку до пріоритету.

Ці збільшення пріоритету не вічні. Вони негайно набувають чинності, але якщо потік використовує повністю свій наступний квант, він втрачає один пункт пріоритету, якщо він використовує ще квант, то переміщається ще на рівень нижче і т. д. аж до базового рівня.

Остання модифікація алгоритму планування W2K полягає в тому, що коли вікно стає вікном переднього плану, усі його потоки отримують триваліші кванти часу (величина надбавки зберігається в системному реєстрі).

3.7.4 Взаємовиключення

У багатозадачних однопроцесорних системах процеси чергуються, забезпечуючи ефективне виконання програм. У багатопроцесорних системах можливе не тільки чергування, а й перекриття процесів.

Способи взаємодії процесів (потоків) можна класифікувати за ступенем обізнаності одного процесу про існування іншого.

1. Процеси *не обізнані* про наявність один одного (наприклад, процеси різних завдань одного або різних користувачів). Це незалежні процеси, не призначені для спільної роботи. Хоча ці процеси і не працюють спільно, ОС повинна вирішувати питання конкурентного використання ресурсів. Наприклад, два незалежні додатки можуть вимагати доступ до одного й того самого диска або принтера. ОС має регулювати такі звернення.

2. Процеси *побічно обізнані* про наявність один одного (наприклад, процеси одного завдання). Ці процеси не обов'язково мають бути обізнані про наявність один одного з точністю до ідентифікатора процесу, однак вони розділяють доступ до деякого об'єкта, наприклад буфера введення-виведення, до файлу або БД. Такі процеси демонструють співпрацю під час поділу спільного об'єкта.

3. Процеси *безпосередньо обізнані* про наявність один одного (наприклад, процеси, що працюють послідовно або по черзі в рамках одного завдання). Такі процеси здатні спілкуватися один з одним з використанням ідентифікаторів процесів і спочатку створені для спільної роботи. Ці

процеси також демонструють співпрацю під час роботи.

Спільне використання спільного ресурсу різними завданнями (або процесами) за відсутності узгодженості може призвести до помилок функціонування. Це завдання розв'язують ***шляхом взаємовиключення***, тобто шляхом надання кожному процесу монопольного виняткового права доступу до даних, що розділяються.

Взаємовиключення необхідне тільки в тому випадку, коли процеси звертаються ***до загальних даних***.

Якщо вони виконують операції, що не призводять до конфліктних ситуацій, вони повинні працювати ***паралельно***.

Коли процес здійснює звернення до даних, що розділяються, то кажуть, що він перебуває ***на критичній ділянці***. Таким чином, розв'язання проблеми взаємовиключення в тому, що якщо один процес перебуває на своїй критичній ділянці, необхідно виключити входження іншого процесу на свою критичну ділянку. Виконання іншого процесу триває, але без входу в критичну ділянку (рис. 11). Якщо ж це неможливо, ***процес має очікувати звільнення критичних даних***. Це одна з ключових проблем паралельного програмування. Вона розв'язується програмно, а в особливо відповідальних випадках апаратно.

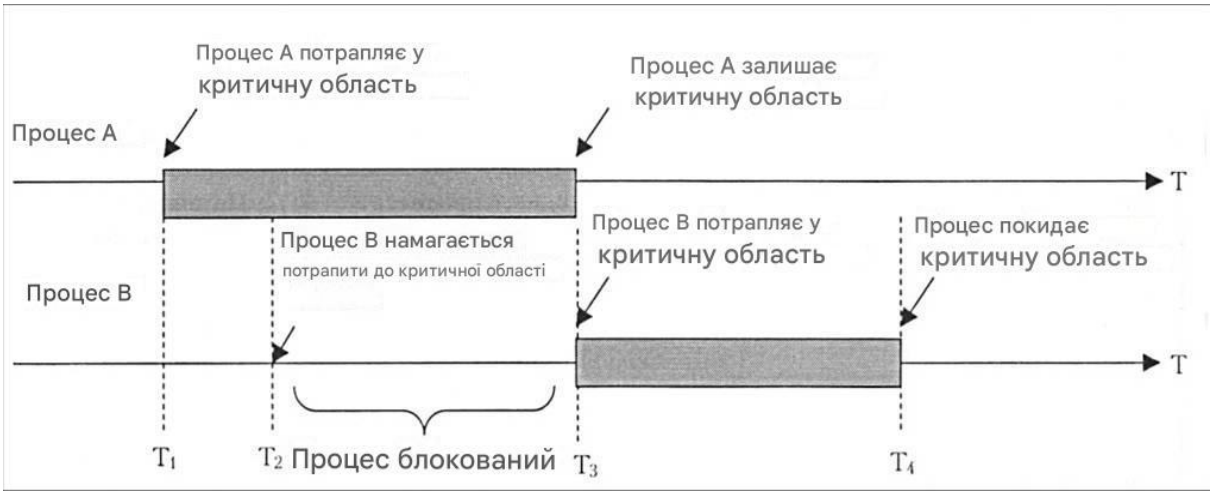

Рис. 11− Поведінка процесів у критичній області

Вимоги до критичної ділянки:

- у будь-який момент часу тільки один процес може перебувати всередині критичної ділянки;
- жоден процес не може залишатися всередині критичної ділянки нескінченно довго;
- жоден процес не повинен чекати нескінченно довго входу в критичну ділянку.

Для двох процесів завдання вирішується алгоритмом Деккера. У цьому

алгоритмі два завдання конкурують за використання спільної критичної ділянки. Доступ до критичної ділянки надається завданням поперемінно. Цей алгоритм був удосконалений Петерсоном.

```
shared int ready[2] = {0, 0};
спільний int turn;
while (some condition)&&
 {
   ready [i] = 1;
   оборот =1- i;
  while (ready [1-i]&& turn == 1-i);
   критичний розділ готовий [i] 0;=
 розділ залишків
}
```

Під час виконання прологу критичної секції процес Pi заявляє про свою готовність виконати критичну ділянку і одночасно пропонує іншому процесу приступити до її виконання. Якщо обидва процеси підійшли до прологу практично одночасно, то вони обидва оголосять про свою готовність і запропонують виконуватися один одному. При цьому одна з пропозицій завжди буде після іншої. Тим самим роботу в критичній ділянці продовжить процес, якому було зроблено останню пропозицію.

Для трьох або більше процесів розроблено так званий "алгоритм булочної" (bakery algorithm). Основна його ідея виглядає так. Кожен новоприбулий клієнт (він же процес) отримує "талончик" на обслуговування з номером. Клієнт із найменшим номером на "талончику" обслуговується наступним. На жаль, через специфіку операції обчислення наступного номера алгоритм булочної не гарантує, що у всіх процесів будуть "талончики" з різними номерами. У разі рівності номерів на "талончиках" у двох або більше клієнтів першим обслуговується клієнт із меншим значенням імені (імена можна порівнювати в лексикографічному порядку).

Структури даних, що розділяються, для алгоритму - це два масиви:

```
shared enum {false, true} choosing[n];
спільний int number[n];
```

Спочатку елементи цих масивів ініціюються значеннями false і 0 відповідно. Уведемо такі позначення:

(a,b) < (c,d), якщо a < c або якщо a == c і b < d

max(a0, al,...,an) - це число k таке, що k >= ai для всіх i = 0, ...,n.

Структура процесу Pi для алгоритму булочної наведена нижче

```
while (деяка умова):
{
  choosing[i] = true;
```

```
 number[i] = max (number[0],number[n-1]) + 1;
 choosing[i] = false;
 for(j = 0; j<n; j++)
 {
  while (choosing[i]);
  while (number[j] != 0 && (number[j],j) < (number[i],i));
 }
критичний розділ
number[i ] = 0;
розділ залишків
}
```

3.7.5 Семафори

Одним із перших механізмів, запропонованих для синхронізації поведінки процесів, стали семафори, концепцію яких описав Дейкстра ще 1965 року.

Механізм взаємовиключення може бути реалізовано за допомогою семафорів: двійкових і лічильних.

Із семафором пов'язані ініціалізація і дві операції:

P(s) - закриття,

V(s) - відкриття.

Двійковий семафор має два стани - 0 і 1. Закритий семафор перебуває у стані 0, відкритий - у стані 1.

Під час ініціалізації семафора визначається його початковий стан:

```
ІніціалізаціяСемафора(s, 1);
```

або

```
ІніціалізаціяСемафора(s, 0);
```

Операція **P(s)**:

```
if s = 1
  then s := 0 { закрити семафор }
  else блокувати процес, що звернувся за s;
    встановити на виконання готовий процес;
```

Операція **V(s)**:

```
if список процесів, які очікують s, не порожній
  then деблокувати процес, який очікує на S
  else s := 1; { відкрити семафор }
```

Критична ділянка облямована операціями P(s) і V(s). Таким чином, двійковий семафор забезпечує логіку роботи з подією (Event).

Семафори, що рахують, корисні під час виділення одиночного ресурсу із загального пулу.

Під час ініціалізації в s заносять кількісний показник обсягу ресурсів пулу. P(s) викликає зменшення лічильника ресурсу на одиницю, а V(s) - збільшення лічильника на одиницю, тобто

операція P(s):
якщо s > 0
 тоді s := s - 1
 else очікувати на s;

операція V(s):
if список процесів, які очікують s, не порожній
 then дозволити одному процесу працювати
 else s := s + 1;

Семафори, що рахують, реалізовано в програмному інтерфейсі Windows. Їхнє використання аналогічне використанню подій і критичних секцій, тобто семафор може бути в позначеному або невідзначеному стані.

3.7.6 Тупики

Кажуть, що в мультипрограмній системі процес перебуває *у стані глухого кута*, якщо він очікує події, яка ніколи не відбудеться. "Зависання" системи - це ситуація, коли один або більше процесів опиняються в стані глухого кута.

Приклад. В операційних системах глухі кути здебільшого виникають унаслідок *звичайної конкуренції за володіння ресурсами, які виділяють або закріплюють* (тобто ресурсами, які в кожний момент часу відводять тільки одному користувачеві, і які тому іноді називають ресурсами послідовного використання).

Нехай Процес1 утримує Ресурс1 і очікує звільнення Ресурсу2. Процес2 утримує Ресурс2 і очікує звільнення Ресурсу1. Кожен процес чекає, щоб інший процес звільнив потрібний йому ресурс, до того ж кожен не звільняє свій ресурс доти, доки інший не звільнить свій ресурс тощо. Такий стан кругового очікування характерний для систем у тупиковому стані (рис. 12).

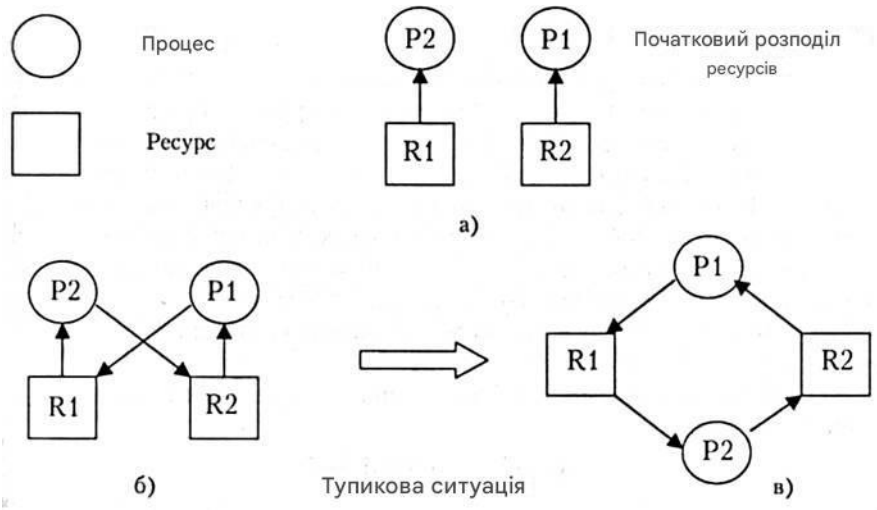

Рис. 12 − Приклад тупикової ситуації

Можна сформулювати чотири необхідні умови глухого кута.

Умова взаємовиключення: процеси вимагають надання їм права монопольного управління ресурсами, які їм виділяють.

Умова очікування ресурсів: процеси утримують ресурси, уже виділені їм, очікуючи виділення додаткових ресурсів.

Умова неперерозподілюваності: ресурси не можна відібрати у процесів, які утримують їх, поки ці ресурси не будуть використані для завершення роботи.

Умова кругового очікування: існує кільцевий ланцюг процесів, які утримують ресурси, що потрібні наступному процесу ланцюга.

У дослідженнях із проблеми глухих кутів можна виокремити такі *основні напрями*:

- запобігання глухим кутам;
- обхід тупиків;
- виявлення тупиків;
- відновлення після глухих кутів.

Запобігання глухим кутам. При запобіганні тупиків метою є забезпечення умов, що виключають можливість виникнення тупикових ситуацій. Такий підхід є цілком коректним рішенням у тому, що стосується самого глухого кута, однак він часто призводить до нераціонального використання ресурсів. Встановлено, що безвихідь буде неможливою, якщо буде порушено хоча б одну із зазначених вище необхідних умов.

Пропонується стратегія, що включає три принципи:

- кожен процес має запитувати всі необхідні ресурси одразу і не починатися, поки їх не буде виділено;
- якщо процес вимагає додаткових ресурсів і отримує відмову, він повинен звільнити всі ресурси і запросити потім всю сукупність ресурсів;
- введення лінійної впорядкованості за типами ресурсів.

Усі елементи цієї стратегії мають свої недоліки.

Обхід тупиків. Використовуються алгоритми, що дають змогу обійти небезпечні ділянки під час розподілу ресурсів (алгоритми Деккера, Пітерсона, Дейкстри).

Виявлення глухого кута - це встановлення факту, що виникла безвихідна ситуація, і визначення процесів і ресурсів, залучених у безвихідну ситуацію.

Алгоритми виявлення глухих кутів, як правило, застосовують у системах, де виконуються перші три необхідні умови виникнення тупикової ситуації. Ці алгоритми потім визначають, чи не створився режим кругового очікування. Таким чином, виявлення глухого кута зводиться до перевірки наявності циклічних запитів на ресурси. Можна включати в ОС алгоритми виявлення, а можна покластися на оператора комп'ютера.

Відновлення після глухого кута - систему, що опинилася в глухому куті, необхідно вивести з нього, порушивши одну або кілька умов його існування. Зазвичай це виконується шляхом примусового виведення деякого процесу із системи, щоб можна було використовувати його ресурси.

Найдоцільніший спосіб - це ефективний механізм призупинення/відновлення процесу.

Інший спосіб - створення контрольних точок (дампів пам'яті) і перезапуск.

Контрольні питання

1. Що таке процес в операційній системі, і які компоненти він включає?
2. Які основні завдання операційної системи при управлінні процесами?
3. Як операційна система використовує таблиці для управління пам'яттю, пристроями введення-виведення, файлами та процесами?
4. У чому полягає відмінність між процесом і потоком, і як ОС розподіляє ресурси між ними?
5. Що таке багатозадачність (мультипрограмування), і як вона впливає на ефективність роботи ОС?
6. Які особливості мультипроцесорних систем і як вони відрізняються від мультипрограмних систем?
7. Що таке симетричне та асиметричне мультипроцесування, і як вони впливають на архітектуру обчислювальних систем?
1. Що таке пріоритетне обслуговування потоків у контексті витісняючих алгоритмів планування?
2. Як ОС визначає пріоритет процесу та на яких підставах це відбувається?
7. Які основні вимоги до критичної ділянки в системах

багатозадачності?

8. Що таке "алгоритм булочної" і як він працює у разі конфлікту доступу кількох процесів до спільного ресурсу?

9. Що таке семафор і які типи семафорів використовуються для синхронізації процесів?

10. Як двійкові та лічильні семафори забезпечують взаємовиключення під час доступу до ресурсів?

11. Що таке стан глухого кута в операційній системі і як він виникає?

12. Які підходи використовуються для запобігання тупикам у мультипрограмних системах?

4 УПРАВЛІННЯ ПАМ'ЯТТЮ

4.1 Функції ОС з управління пам'яттю

Під пам'яттю (memory) мається на увазі оперативна пам'ять комп'ютера. В однозадачних операційних системах основна пам'ять поділяється на дві частини. Одна частина - для операційної системи (резидентний монітор, ядро), а друга - для програми, що виконується в поточний момент часу. У багатопрограмних ОС "призначена для користувача" частина пам'яті - найважливіший ресурс обчислювальної системи - має бути розподілена для розміщення декількох процесів, зокрема і процесів ОС. Це завдання розподілу виконується операційною системою динамічно спеціальною підсистемою управління пам'яттю (memory management). Ефективне управління пам'яттю життєво важливе для багатозадачних систем. Якщо в пам'яті перебуватиме невелика кількість процесів, то значну частину часу процеси перебуватимуть у стані очікування вводу-виводу і завантаження процесора буде низьким.

З появою мультипрограмування завдання ОС, пов'язані з розподілом наявної пам'яті між кількома програмами, що одночасно виконуються, істотно ускладнилися.

Функціями ОС з управління пам'яттю в мультипрограмних системах є:

- відстеження (облік) вільної та зайнятої пам'яті;
- початкове і динамічне виділення пам'яті процесам додатків і самій операційній системі та звільнення пам'яті після завершення процесів;
- налаштування адрес програми на конкретну область фізичної пам'яті;
- повне або часткове витіснення кодів і даних процесів з ВП на диск, коли розміри ВП недостатні для розміщення всіх процесів, і повернення їх у ВП;
- захист пам'яті, виділеної процесу від можливих втручань з боку інших процесів;

- дефрагментація пам'яті.

Перераховані функції особливого пояснення не потребують, окрім завдання перетворення адрес програми під час її завантаження в ОП.

Для ідентифікації змінних і команд на різних етапах життєвого циклу програми використовують *символьні* імена, *віртуальні* (математичні, умовні, логічні - усе це синоніми) і *фізичні* адреси (рис. 13).

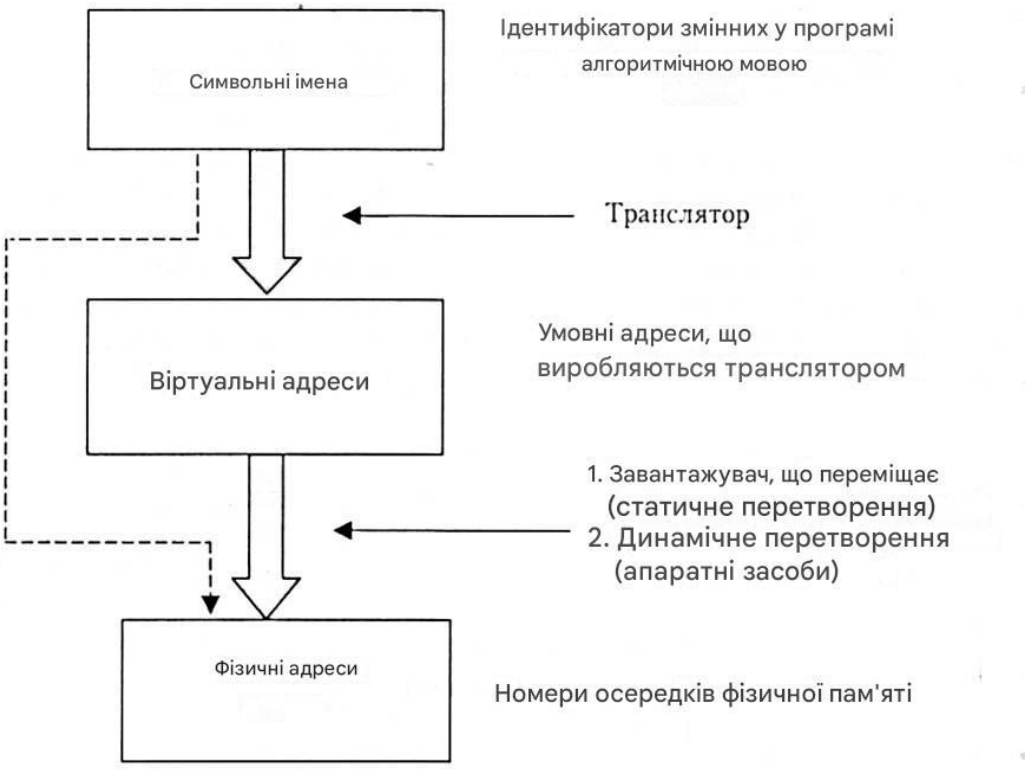

Рис. 13 − Порядок перетворення адрес програми

Символьні імена присвоює користувач під час написання програм алгоритмічною мовою або асемблером. Віртуальні адреси виробляє транслятор, який перекладає програму машинною мовою. Оскільки під час трансляції невідомо, у яке місце оперативної пам'яті буде завантажено програму, то транслятор присвоює змінним і командам віртуальні (умовні) адреси, вважаючи за умовчанням, що початковою адресою програми буде нульова адреса.

Фізичні адреси відповідають номерам комірок оперативної пам'яті, де насправді будуть розташовані змінні та команди.

Існує два принципово відмінні підходи до перетворення віртуальних адрес у фізичні.

У першому випадку таке перетворення виконується один раз для кожного процесу під час початкового завантаження програми в пам'ять. Перетворення здійснює завантажувач, що переміщає, на підставі наявних у нього даних про початкову адресу фізичної пам'яті, у яку належить

завантажувати програму, а також інформації, що надається транслятором про адресно-залежні елементи програми.

Другий спосіб полягає в тому, що програма завантажується в пам'ять у віртуальних адресах. Під час виконання програми при кожному зверненні до пам'яті операційна система перетворює віртуальні адреси на фізичні.

4.2 Класифікація методів розподілу пам'яті

Існують різні підходи до класифікації пам'яті. У цьому випадку використовується класифікація методів розподілу пам'яті, у якій виділено два класи методів - з переміщенням сегментів процесів між ОП і ВП (диском) і без переміщення, тобто без залучення зовнішньої (віртуальної) пам'яті (рис. 14). Ця класифікація враховує тільки основні ознаки методів. Для кожного методу може бути використано кілька різних алгоритмів його реалізації.

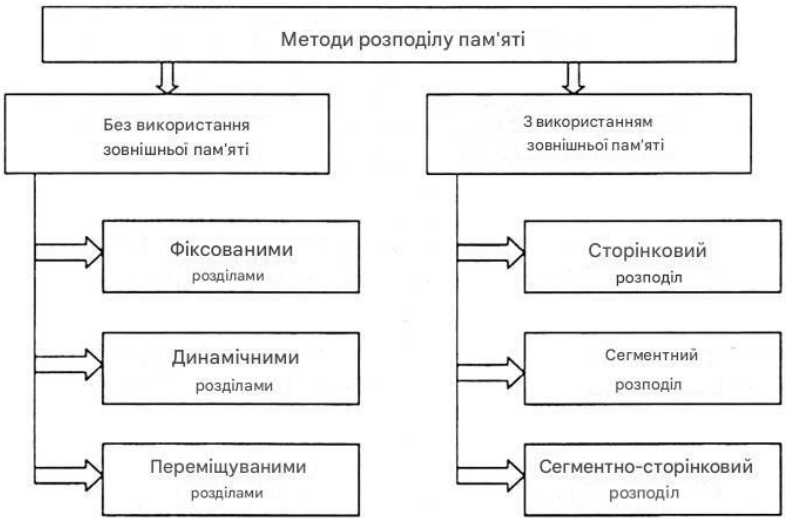

Рис. 14 − Класифікація методів віртуальної пам'яті

4.3 Розподіл пам'яті без використання зовнішньої пам'яті

Існує два варіанти **фіксованого розподілу пам'яті**. Одна можливість полягає у використанні розділів однакового розміру. У цьому разі будь-який процес, розмір якого не перевищує розмір розділу, може бути завантажений у будь-який доступний розділ. Якщо всі розділи зайняті і немає жодного процесу в стані готовності або роботи, ОС може вивантажити процес із будь-якого розділу і завантажити інший процес, забезпечуючи тим самим процесор роботою (рис. 15).

Під час використання розділів з однаковим розміром є дві труднощі.

1. Програма може бути занадто великою для розміщення в розділі. У цьому випадку програміст повинен розробляти програму, що використовує

оверлеї (overlays), щоб у будь-який момент часу був потрібен тільки один розділ пам'яті. Коли потрібен модуль, відсутній у цей момент в ОП, користувацька програма повинна сама його завантажити в розділ пам'яті програми. Таким чином, у цьому разі керування пам'яттю багато в чому покладається на програміста.

2. використання ОП вкрай неефективне. Будь-яка програма, незалежно від її розміру, займає розділ цілком. При цьому можуть залишатися невикористані ділянки пам'яті великого розміру. Цей феномен появи невикористаної пам'яті називається внутрішньою *фрагментацією* (internal fragmentation).

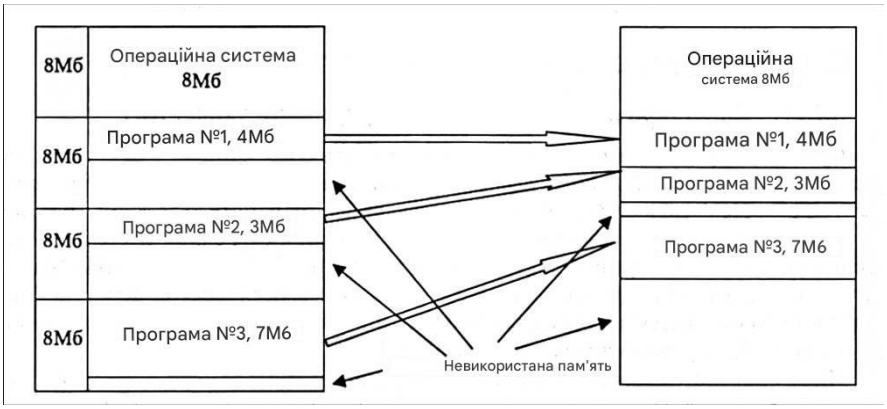

Рис. 15 − Розподіл пам'яті фіксованими розділами

Боротися з цими труднощами (хоча й не усунути їх повністю) можна за допомогою використання розділів різних розмірів. У цьому разі програма великих розмірів може обійтися без оверлеїв, а розділи малого розміру дають змогу зменшити внутрішню фрагментацію під час завантаження невеликих програм.

Коли розділи мають різні розміри, є два можливих підходи до призначення процесів розділам пам'яті. Найпростіший шлях полягає в тому, щоб кожен процес розміщувався в найменшому розділі, здатному вмістити цей процес. За такого підходу для кожного розділу потрібна черга планувальника, у якій зберігаються вивантажені з пам'яті процеси, призначені для цього розділу пам'яті. Перевага такого способу в можливості розподілу процесів між розділами ОП, щоб мінімізувати внутрішню фрагментацію.

Недолік полягає в тому, що окремі черги для розділів можуть призвести до неоптимальності розподілу пам'яті для системи в цілому. Наприклад, якщо в деякий момент часу немає жодного процесу розміром від 7 до 12 Мбайт, то розділ розміром 12 Мбайт пустуватиме.

Тому кращим є використання однієї черги для всіх процесів. У момент, коли потрібно завантажити процес у ВП, вибирається найменший доступний розділ, здатний вмістити цей процес.

Однак у цих схем є серйозні недоліки.

1. Кількість розділів, визначена в момент генерації системи, обмежує кількість активних процесів (тобто рівень багатозадачності).

2. Оскільки розміри розділів встановлюються заздалегідь під час генерації системи, невеликі завдання призводять до неефективного використання пам'яті.

При **динамічному розподілі пам'яті** утворюється змінна кількість розділів різної мінливої довжини. Під час розміщення процесу в основній пам'яті для нього виділяється строго необхідна кількість пам'яті. Як приклад розглянемо використання 64 Мбайт (Рис. 16) основної пам'яті.

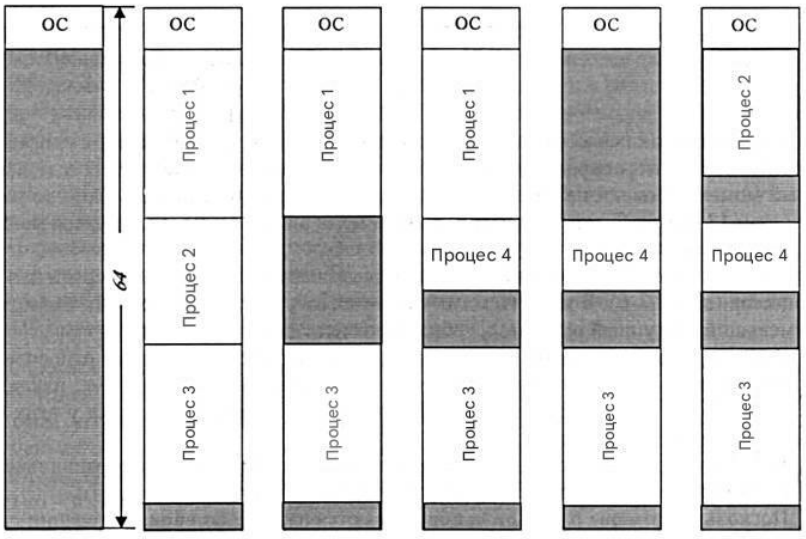

Рис. 16 − Приклад розподілу пам'яті динамічними розділами

Спочатку вся пам'ять порожня, за винятком області, використовуваної ОС. Перші три процеси завантажуються в пам'ять, починаючи з адреси, де закінчується ОС, і використовують стільки пам'яті, скільки потрібно даному процесу. Після цього в кінці ОП залишається вільна ділянка пам'яті, занадто мала для розміщення четвертого процесу. У деякий момент часу всі процеси в пам'яті виявляються неактивними, і операційна система вивантажує другий процес, після чого залишається достатньо пам'яті для завантаження нового, четвертого процесу.

Оскільки процес 4 менший за процес 2, з'являється ще вільна ділянка пам'яті. Після того як у деякий момент часу всі процеси виявилися неактивними, став готовим до роботи процес 2, але вільного місця в пам'яті для нього не знаходиться. Тому система змушена вивантажити процес 1, щоб звільнити необхідне місце і розмістити процес 2 в ОП. Як показує цей приклад, цей метод добре починає роботу, але погано продовжує. Зрештою, він призводить до наявності безлічі дрібних вільних ділянок пам'яті, в яких

немає можливості розмістити будь-який новий процес. Це явище називається зовнішньою фрагментацією (external fragmentation), що відображає той факт, що сильно фрагментованою стає пам'ять, зовнішня по відношенню до всіх розділів.

Динамічний розподіл пам'яті переміщуваними розділами - один із методів подолання зовнішньої фрагментації (ущільнення (compaction) процесів в ОП). Здійснюється це переміщенням усіх зайнятих ділянок так, щоб вільна пам'ять утворила єдину вільну область. На додаток до функцій, які ОС виконує під час розподілу пам'яті динамічними розділами, у даному випадку вона повинна ще час від часу копіювати вміст розділів з одного місця в інше, коригуючи таблиці вільних і зайнятих областей. Цей процес називається ущільненням або стисненням (рис. 17).

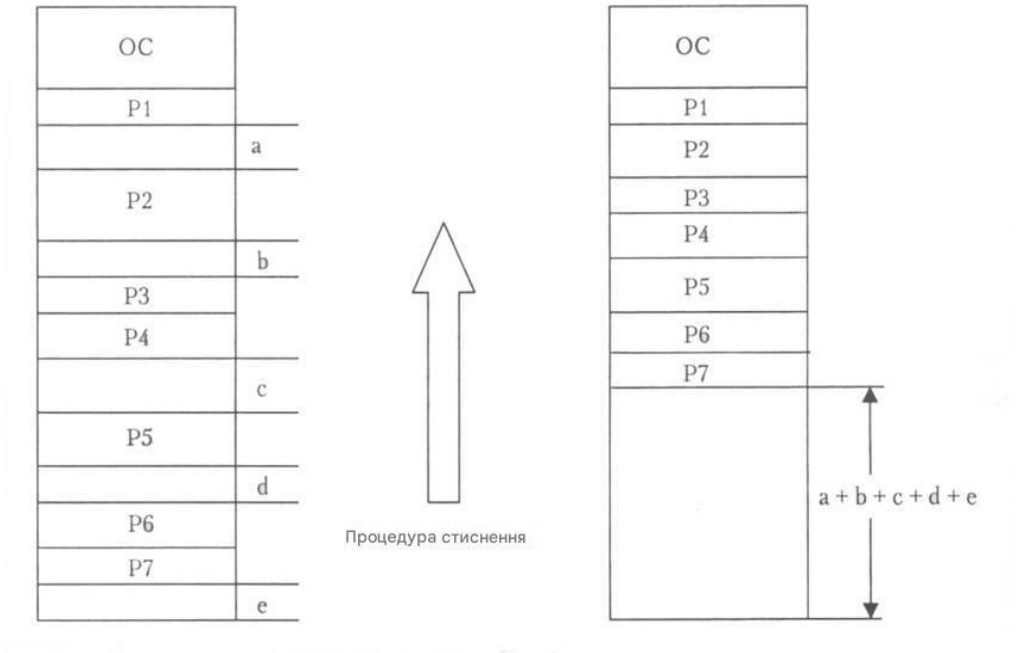

Рис. 17 − Динамічний розподіл пам'яті між розділами

Функції операційної системи з управління пам'яттю в цьому випадку.

1. Переміщення всіх зайнятих ділянок у бік старших або молодших адрес під час кожного завершення процесу або для новостворюваного процесу в разі відсутності розділу достатнього розміру.

2. Корекція таблиць вільних і зайнятих областей.

3. зміна адрес команд і даних, до яких звертаються процеси під час їх переміщення 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.

Ущільнення може виконуватися або при кожному завершенні процесу,

або тільки тоді, коли для новостворюваного процесу немає вільного розділу достатнього розміру. У першому випадку потрібно менше обчислювальної роботи під час коригування таблиць вільних і зайнятих областей, а в другому - рідше виконується процедура стиснення.

Перевагою розподілу пам'яті переміщуваними розділами є ефективне використання оперативної пам'яті, виняток внутрішньої і зовнішньої фрагментації, недоліком - додаткові накладні та часові витрати на реалізацію в ОС механізму ущільнення.

Для адресації команд і даних завантажуваної програми використовується динамічне перетворення адрес.

Оскільки розміщення команд і даних, до яких звертається процес, не є фіксованим, у 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.

Спосіб трансляції, який зазвичай використовують, показано на рис. 18.

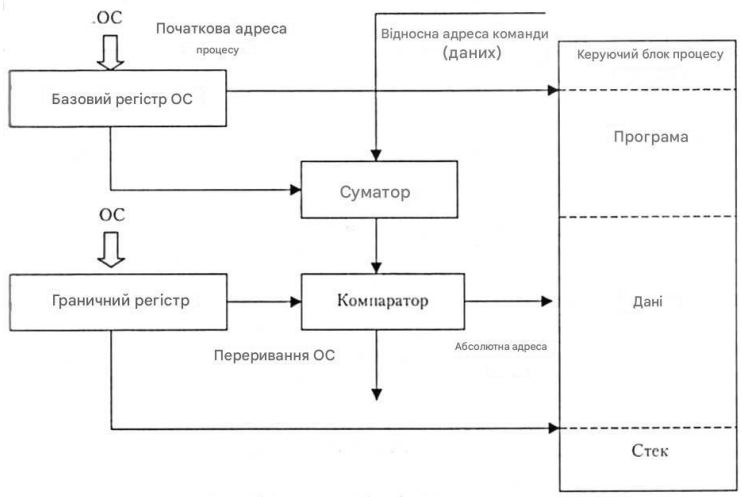

Рис. 18 − Схема перетворення відносної в абсолютний

Коли процес переходить у стан виконання, у спеціальний регістр, який називають базовим, завантажується початкова адреса процесу в основній пам'яті. Крім того, використовується граничний регістр, у якому міститься адреса останньої клітинки програми. Під час виконання процесу відносні адреси в командах обробляються процесором у два етапи. Спочатку до відносної адреси додається значення базового регістра для отримання абсолютної адреси. Потім отримана абсолютна адреса порівнюється зі значенням у граничному регістрі. Якщо отримана абсолютна адреса належить цьому процесу, команда може бути виконана. В іншому разі генерується відповідне даній помилці переривання.

4.4 Методи структуризації віртуальної пам'яті

Більшість систем віртуальної пам'яті використовують техніку, звану сторінковою організацією пам'яті. Будь-який процес, реалізований у комп'ютері, може звернутися до безлічі адрес у пам'яті. Адреси можуть формуватися з використанням індексації, базових регістрів, сегментних регістрів та іншими шляхами. Ці програмно формовані адреси, звані віртуальними адресами, формують віртуальний адресний простір.

Уся наявна нині безліч реалізацій віртуальної пам'яті розрізняється в основному способом структуризації віртуального адресного простору.

Нині виділяють три методи реалізації віртуальної пам'яті:

1) сторінкова віртуальна пам'ять організовує переміщення даних між основною пам'яттю і диском сторінками - частинами віртуального адресного простору фіксованого і порівняно невеликого розміру;

2) сегментна віртуальна пам'ять передбачає переміщення даних сегментами - частинами віртуального адресного простору довільного розміру, отриманими з урахуванням смислового значення даних;

3) сегментно-сторінкова віртуальна пам'ять використовує дворівневий розподіл: віртуальний адресний простір ділиться на сегменти, а потім сегменти діляться на сторінки. Одиницею переміщення даних є сторінка.

Для тимчасового зберігання сегментів і сторінок на диску відводиться спеціальна область або спеціальний файл (сторінковий файл або файл підкачки - paging file). Поточний розмір сторінкового файлу є важливим параметром, що впливає на можливості операційної системи: що більший сторінковий файл, то більше додатків може одночасно виконувати ОС (за фіксованого розміру оперативної пам'яті). Однак необхідно розуміти, що збільшення кількості додатків, які одночасно працюють, за рахунок збільшення розміру сторінкового файлу уповільнює їхню роботу, оскільки значна частина часу при цьому витрачається на переміщення даних на диск і назад.

4.4.1 Сторінкова організація віртуальної пам'яті

У разі сторінкової організації віртуальний адресний простір кожного процесу ділиться на частини однакового, фіксованого для даної системи розміру, які називаються віртуальними сторінками (Virtual pages). У загальному випадку розмір віртуального адресного простору не кратний розміру сторінки, тому остання сторінка доповнюється фіксованою областю.

Уся оперативна пам'ять машини також ділиться на частини такого ж розміру, які називаються фізичними сторінками (або блоками, або кадрами). Розмір сторінки вибирають рівним ступеню двійки: 1024, 2408, 4096 байт тощо. Це дає змогу спростити механізм перетворення адрес.

Під час створення процесу ОС завантажує в операційну пам'ять кілька його віртуальних сторінок (початкові сторінки кодового сегмента і сегмента

даних). Копія віртуального адресного простору процесу знаходиться на диску. Суміжні віртуальні сторінки не обов'язково знаходяться в суміжних фізичних сторінках. Для кожного процесу ОС створює таблицю сторінок - інформаційну структуру, що створює записи про всі віртуальні сторінки процесу (рис. 19).

Запис таблиці (дескриптор сторінки) містить таку інформацію:

- номер фізичної сторінки (N ф.с.), у яку завантажено цю віртуальну сторінку;
- ознака присутності P, що встановлюється в одиницю, якщо ця сторінка перебуває в оперативній пам'яті;
- ознака модифікації сторінки D, яку встановлюють в одиницю щоразу, коли здійснюють запис за адресою, що належить до цієї сторінки;
- ознака звернення A до сторінки, звана також бітом доступу, яку встановлюють в одиницю під час кожного звернення за адресою, що належить до цієї сторінки;
- інші керуючі біти, що слугують, наприклад, для цілей захисту або спільного використання пам'яті на рівні сторінок.

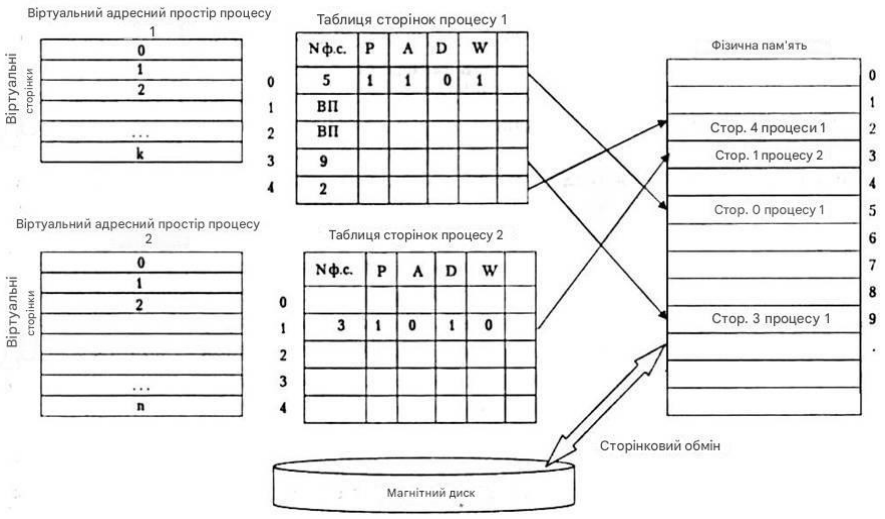

Рис. 19 − Сторінкова організація віртуальної пам'яті

Перелічені ознаки в більшості моделей процесів встановлюються апаратно схемами процесора при виконанні операцій з пам'яттю. Інформація з таблиці сторінок використовується для вирішення питання про необхідність переміщення тієї чи іншої сторінки між пам'яттю і диском, а також для перетворення віртуальної адреси у фізичну. Самі таблиці сторінок, так само як і описувані ними сторінки розміщуються в оперативній пам'яті.

У деяких схемах віртуальної пам'яті при великій кількості записів у таблицях сторінок вони частково зберігаються не в реальній, а у віртуальній пам'яті. Це означає, що самі таблиці сторінок стають об'єктами сторінкової

організації. Але під час роботи процесу щонайменше частина його таблиці сторінок має розташовуватися в основній пам'яті, зокрема запис про сторінку, що виконується зараз.

Віртуальну адресу при сторінковому розподілі можна подати у вигляді пари (P, S_v), де P - номер віртуальної сторінки процесу (нумерація сторінок починається з 0), а S_v - зміщення в межах віртуальної сторінки (рис. 20). 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. Завдання підсистеми віртуальної пам'яті полягає у відображенні пари значень (P, S_v) у пару (N, S_f).

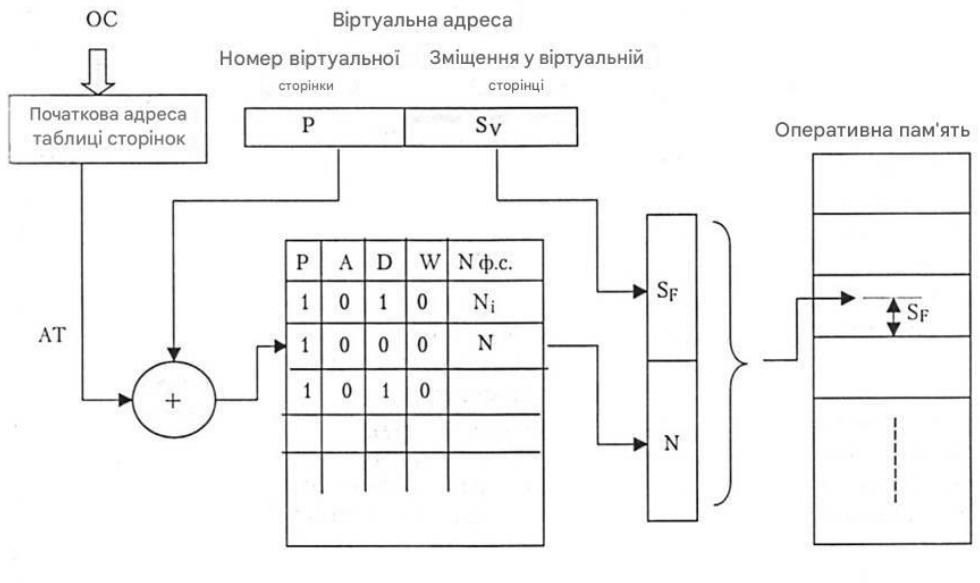

Рис. 20 − Схема перетворення адреси у віртуальній пам'яті

Щоб зрозуміти механізм реалізації цього відображення, слід зупинитися на двох базисних властивостях сторінкової організації. Як уже зазначалося, обсяг сторінок як віртуальної, так і фізичної вибирають рівним ступеню двійки - 2^K (k = 8 і більше) Звідси випливає, що зсув S_v і S_t- можна одержати відокремленням k молодших розділів у двійковому записі віртуальної і відповідно фізичної адреси сторінок. При цьому старші розділи адреси, що залишилися, являють собою двійковий запис номерів віртуальної і відповідно фізичної сторінки. Доповнивши ці номери до нулів, можна отримати початкову адресу віртуальної та фізичної сторінок.

Друга властивість - лінійність адресного простору віртуальної та фізичної сторінки - призводить до того, що $S_f = S_v$. Звідси випливає проста схема перетворення віртуальної адреси у фізичну.

При зверненні до пам'яті за деякою віртуальною адресою (P, Sv) апаратні схеми процесора виконують такі дії.

1 Зі спеціального регістра процесора витягується початкова адреса AT

таблиці сторінок активного процесу. За допомогою суматора S за значеннями AT, P, L (довжина окремого запису в таблиці сторінок) визначається адреса потрібного запису в таблиці сторінок:

A = AT + (P * L).

2. Зчитується номер відповідної фізичної сторінки - N.

3. До номера фізичної сторінки приєднується зміщення S_v.

У підсумку отримана фізична адреса оперативної пам'яті представляється парою значень (N, S_f).

Розглянемо приклад, що пояснює основні характеристики організації сторінок віртуальної пам'яті.

Нехай комп'ютер має оперативну пам'ять об'ємом $E_{оп}$ = 256 Мбайт, розмір сторінки обрано таким, що дорівнює $E_{стр}$= 4 Кбайт.

У цьому випадку кількість фізичних сторінок дорівнює

$N_f = E_{оп} / E_{стор} = 256*20^{20} / 4*2^{10} = 64{,}000$ сторінок.

Для відображення фізичної адреси довільного байта оперативної пам'яті потрібно К = $\log_2 (256*20^{20})$ - 28 двійкових розрядів.

Кількість розрядів для відображення зміщення в сторінці M = $\log_2$ 4 Кбайт = $\log_2$ 4096 = 12.

Якщо процесор має 32-розрядну структуру, то на номер віртуальної країни відводиться 32 - 12 = 20 двійкових розрядів.

Таким чином, кількість віртуальних країн дорівнює $N_B = 2^{20}$ (приблизно 1 млн. віртуальних сторінок).

Нині відомо кілька методів підвищення ефективності функціонування сторінкової віртуальної пам'яті. До них належать такі методи.

1 Складніша структуризація віртуального адресного простору, наприклад дворівнева (типова для 32-бітової адресації).

2 Використання спеціального високошвидкісного кешу для зберігання частини записів таблиці сторінок, який зазвичай називають буфером швидкого перетворення адреси, або буфером пошуку трансляції (translation lookaside buffer - TLB).

3. Вибір оптимального розміру сторінки віртуальної пам'яті.

4 Ефективне управління сторінковим обміном.

4.4.2 Сегментна організація віртуальної пам'яті

При сторінковій організації віртуальний адресний простір ділиться на рівні частини механічно без урахування смислового значення даних. Для багатьох завдань наявність двох і більше окремих віртуальних адресних просторів може виявитися кращим варіантом, ніж один простір. До того ж існує необхідність створення відносно незалежних адресних просторів різної довжини. Це завдання може бути вирішене шляхом сегментної організації пам'яті.

Кожен сегмент містить лінійну послідовність адрес від 0 до деякого

максимуму.

Різні сегменти можуть бути різної довжини. Ба більше, довжини сегментів можуть змінюватися під час виконання. Оскільки кожен сегмент становить окремий адресний простір, різні сегменти можуть зростати і скорочуватися залежно один від одного.

Щоб визначити адресу в такій сегментованій або двовимірній пам'яті, програма має вказати адресу, яка складається з двох частин: номер сегмента та адреса всередині сегмента.

Максимальний розмір сегмента визначається розрядністю віртуальної адреси, наприклад за 32-розрядного мікропроцесора він дорівнює 2^{32} = 4 Гбайт. При цьому максимально можливий віртуальний адресний простір являє собою набір з N віртуальних сегментів (зауважимо, що спільної для сегментів лінійної віртуальної адреси не існує).

Крім простоти управління структурами даних, що збільшуються або скорочуються, сегментована пам'ять має й інші переваги.

До них належать:

- простота компонування окремо скомпільованих процедур (звернення до початкової точки процедури здійснюється адресою виду (n, 0), де n - номер сегмента);

- легкість забезпечення диференційованого доступу до різних частин програми (наприклад, заборонити звертатися для запису в сегмент програми);

- простота організації спільного використання фрагментів програм різними процесами, наприклад, бібліотеки спільного доступу можуть бути оформлені у вигляді окремого сегмента, який може бути включений у віртуальний адресний простір декількох процесів.

Під час завантаження процесу в оперативну пам'ять поміщається тільки частина його сегментів, повна копія віртуального адресного простору міститься в дисковій пам'яті. Для кожного завантажуваного сегмента ОС підшукує безперервну ділянку вільної пам'яті достатнього розміру. Суміжні у віртуальній пам'яті сегменти можуть займати несуміжні ділянки оперативної пам'яті. Якщо під час виконання процесу відбувається звернення до відсутнього в основній пам'яті сегмента, відбувається переривання. Операційна система в цьому разі працює аналогічно подібному процесу в сторінковій віртуальній пам'яті.

На етапі створення процесу під час завантаження його образу в оперативну пам'ять ОС створює таблицю сегментів процесу, аналогічну таблиці сторінок, у якій для кожного сегмента вказується:

- базова фізична адреса початку сегмента в оперативній пам'яті;

- розмір сегмента;

- правила доступу до сегмента;

- ознаки модифікації, присутності та звернення до цього сегмента, а

також деяка інша інформація.

Механізм перетворення віртуальної адреси за сегментної організації дуже схожий з перетворенням віртуальної адреси за сторінкової організації. Однак факт довільного розміру сегментів призводить до того, що не можна обійтися конкатенацією номера сегмента і зміщення. У цьому разі фізична адреса отримується додаванням базової адреси сегмента, яка визначається за номером сегмента n із сегментної таблиці сегментів, і зміщення S. Схему перетворення віртуальної адреси за сегментної організації пам'яті наведено на рис. 21.

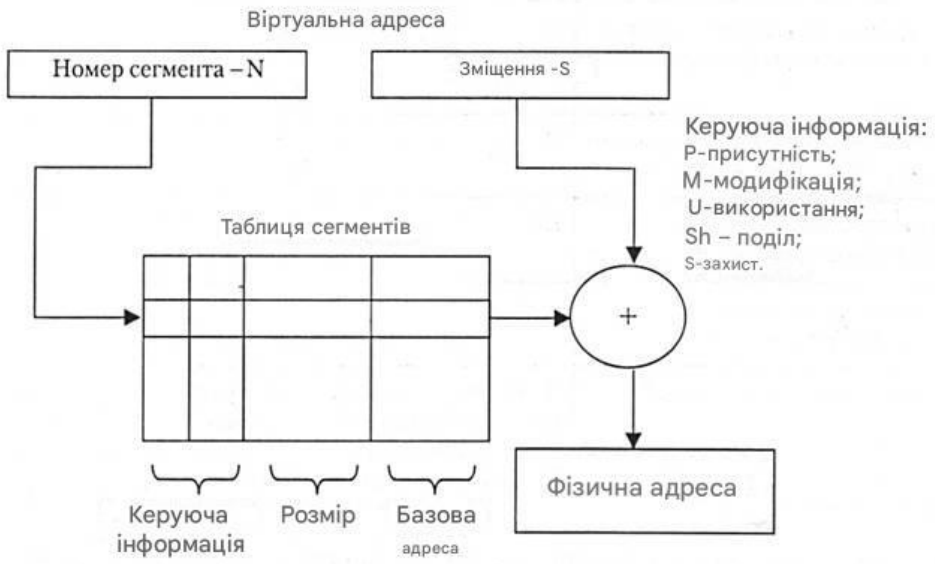

Рис. 21 − Перетворення адреси за сегментної організації пам'яті

Використання операції додавання замість конкатенації уповільнює процедуру утворення віртуальної адреси у фізичну. Іншим недоліком сегментної організації віртуальної пам'яті є велика надлишковість переміщення даних між диском і оперативною пам'яттю, оскільки переміщуються цілком великі сегменти. У багатьох випадках було б достатньо завантажувати і вивантажувати не весь сегмент, а дону або кілька сторінок. Однак найістотніший недолік сегментної організації віртуальної пам'яті - фрагментація пам'яті, яка виникає через довільні розміри сегментів.

4.4.3 Сторінково-сегментна організація пам'яті

Даний метод організації віртуальної пам'яті спрямований на поєднання переваг сторінкового і сегментного методів управління пам'яттю. У такій комбінованій системі адресний простір користувача розбивається на низку сегментів на розсуд програміста. Кожен сегмент зі свого боку розбивається на сторінки фіксованого розміру, що дорівнюють сторінці фізичної пам'яті.

З погляду програміста, логічна адреса в цьому випадку складається з

номера сегмента і зміщення в ньому. З позиції операційної системи зміщення в сегменті слід розглядати як номер сторінки певного сегмента і зміщення в ній (рис. 22).

З кожним процесом пов'язана одна таблиця сегментів і кілька (по одній) на сегмент таблиць сторінок. Під час роботи певного процесу в регістрі процесора зберігається початкова адреса відповідної таблиці сегментів. Отримавши віртуальну адресу, процесор використовує її частину, що представляє номер сегмента, як індекс у таблиці сегментів для пошуку таблиці сторінок цього сегмента. Після цього частина адреси, що являє собою номер сторінки, використовується для пошуку номера фізичної сторінки в таблиці сторінок. Потім частина адреси, що представляє зміщення, використовується для отримання шуканої фізичної адреси шляхом додавання до початкової адреси фізичної сторінки.

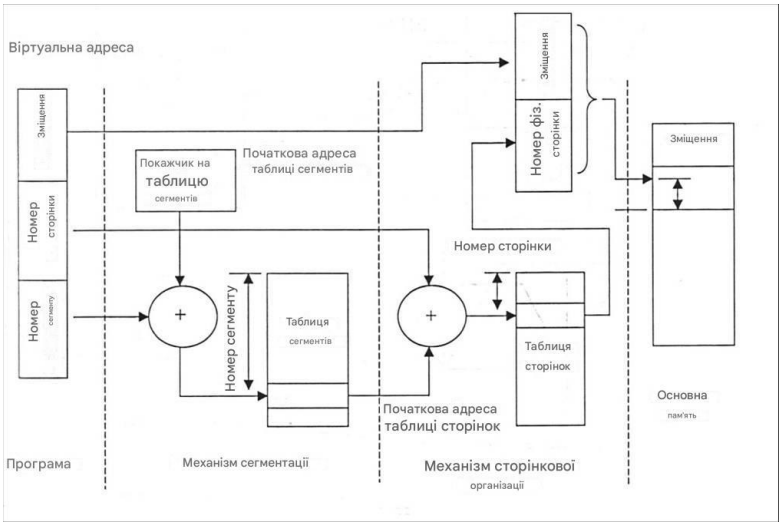

Рис. 22 − Схема перетворення адреси в разі сторінково-сегментної організації пам'яті

Сегментація зручна для реалізації захисту та спільного використання сегментів різними процесами. Оскільки кожен запис таблиці сегментів містить початкову адресу і значення довжини, програма не в змозі ненавмисно звернутися до основної пам'яті за межами сегмента. Для того щоб відрізнити сегменти, що розділяються, від індивідуальних, записи таблиці сегментів містять 1-бітове поле, що має два значення shared (поділюваний) або private (індивідуальний). Для здійснення спільного використання сегмента його поміщають у віртуальний адресний простір кількох процесів, при цьому параметри відображення цього сегмента налаштовують так, щоб вони відповідали одній і тій самій ділянці оперативної пам'яті (роблять це вказівкою однієї і тієї самої базової фізичної адреси сегмента).

Контрольні питання

1. Які основні функції операційної системи щодо управління пам'яттю?
2. Що таке віртуальні та фізичні адреси, і як вони перетворюються?
3. Які існують основні підходи до класифікації методів розподілу пам'яті?
4. Чим відрізняються методи розподілу пам'яті з використанням зовнішньої пам'яті і без неї?
6. Яка основна мета сторінково-сегментної організації пам'яті?
7. Як логічна адреса перетворюється у фізичну адресу в сторінково-сегментній організації пам'яті?
8. Яку роль виконує таблиця сегментів під час перетворення адрес?
9. Яким чином досягається захист пам'яті у сторінково-сегментній організації?
10. Яке значення має поле "shared" або "private" у записах таблиці сегментів?
11. Як реалізується спільне використання сегментів між процесами?
12. Яку інформацію містить кожен запис таблиці сегментів?

5 ФАЙЛОВІ СИСТЕМИ

5.1 Мета та завдання файлової системи

Дані мають зберігатися на пристроях комп'ютерів, що відповідають відповідним вимогам.

1. Пристрої мають давати змогу зберігати дуже великі обсяги даних. До таких пристроїв належать жорсткі магнітні диски, магнітні стрічки, оптично магнітооптичні диски.

2. інформація повинна довгостроково і надійно зберігатися після припинення роботи процесу, що використовує цю інформацію. Довготривалість зберігання забезпечується використанням запам'ятовуючих пристроїв, що не залежать від електроживлення, а висока надійність визначається відповідною організацією операційної системи.

3 Декілька процесів повинні мати можливість отримання одночасного доступу до інформації, тобто має бути забезпечено спільне використання даних.

Розв'язання цих проблем полягає у зберіганні інформації, організованої у файли. Файл - це іменована сукупність даних, що зберігається на будь-якому носії інформації.

Зазвичай єдиним способом роботи з файлами є використання системи управління файлами або інакше - *файлової системи* (ФС). Файлова система

- це частина операційної системи, що включає:

- сукупність усіх файлів на носії інформації (магнітному або оптичному диску, магнітній стрічці тощо);

- набори структур даних, що використовуються для управління файлами (каталоги і дескриптори файлів, таблиці розподілу вільного і зайнятого простору на диску та ін.);

- комплекс системних програмних засобів, що реалізують різні операції над файлами (створення, знищення, читання, запис тощо).

Завдання, розв'язувані файловою системою, багато в чому визначаються способом організації обчислювального процесу (найпростіші - в однопрограмних і однокористувацьких ОС, найскладніші - в мережевих ОС).

У багатозадачних, багатокористувацьких ОС завданнями файлової системи є:

1) відповідність вимогам управління даними та вимогам з боку користувачів, що включають можливість зберігання даних і виконання операцій з ними;

2) гарантування коректності даних, що містяться у файлі; оптимізація продуктивності, як з погляду системи (пропускна спроможність), так і з погляду користувача (час відгуку); підтримка вводу-виводу для різних типів пристроїв зберігання інформації; мінімізація або повне унеможливлення можливих втрат або пошкоджень даних; захист файлів від несанкціонованого доступу;

3) забезпечення підтримки спільного використання файлів кількома користувачами (зокрема засоби блокування файлу і його частин, виключення глухих кутів, узгодження копій тощо);

4) забезпечення стандартизованого набору підпрограм інтерфейсу введення-виведення.

Мінімальним набором вимог до файлової системи з боку користувача діалогової системи загального призначення можна вважати таку сукупність можливостей, що надається користувачеві:

- створення, видалення, читання і зміна файлів;

- контрольований доступ до файлів інших користувачів.

- структурування файлів відповідно до розв'язуваного завдання;

- переміщення даних між файлами;

- резервування та відновлення файлів у разі пошкодження;

- доступ до файлів за символьними іменами.

Файлова система дає змогу програмам обходитися набором доволі простих операцій для виконання дій над деяким абстрактним об'єктом, що представляє файл. При цьому програмістам не потрібно мати справу з деталями дійсного розташування даних на диску, буферизацією даних та іншими низькорівневими проблемами передавання даних із

запам'ятовуючого пристрою. Усі ці функції файлова система бере на себе.

Таким чином, файлова система відіграє роль проміжного шару, що екранізує всі складнощі фізичної організації довготривалого сховища даних і створює для програм простішу логічну модель цього сховища, а потім надає їм набір зручних у використанні команд для маніпулювання файлами.

5.2 Організація файлів і доступ до них

Файлові системи підтримують кілька функціонально різних типів файлів, до числа яких входять звичайні файли, що містять інформацію довільного характеру (текст, графіка, звук тощо), файли-каталоги, спеціальні файли, іменовані конвеєри, файли, які відображають на пам'ять, тощо.

Звичайні файли, або просто файли, або *регулярні файли,* містять інформацію, яку в них заносить користувач або яка утворюється в результаті роботи системних і користувацьких програм. Більшість ОС не контролюють вміст структуру регулярних файлів, які в основному є ASCII-файлами або двійковими файлами. ASCII-файли складаються з текстових рядків. Вони можуть відображатися на екрані та виводитися на друк без будь-якого перетворення, і можуть редагуватися практично будь-яким текстовим редактором. *Двійкові файли* мають певну внутрішню структуру, яка відома програмі, що використовує цей файл.

Каталоги - це системні файли, що забезпечують підтримку структури файлової системи. Вони містять системну довідкову інформацію про набір файлів, згрупованих користувачем за якою-небудь неформальною ознакою (договори, реферати, курсові проекти тощо). У багатьох ОС до каталогу можуть входити інші файли, зокрема інші каталоги, завдяки чому утворюється деревоподібна структура, зручна для пошуку потрібного файлу. Каталоги встановлюють відповідність між іменами файлів та їхніми характеристиками, використовуваними файловою системою для управління файлами. До числа таких характеристик входить тип файлу, права доступу до файлу, його розташування на диску, розмір, дата і час створення та ін.

Спеціальні файли - це фіктивні файли, асоційовані з пристроями введення-виведення, які використовуються для уніфікації механізму доступу до послідовних пристроїв введення-виведення, таких як термінали, принтери тощо. (наприклад MS-DOS розглядає монітор і клавіатуру як файли зі стандартним ім'ям con - консоль, а принтер - як файл prn). Блокові спеціальні файли використовуються для моделювання дисків.

Іменовані контейнери (канали) являють собою циклічні буфери, що дають змогу вихідний файл однієї програми з'єднати із вхідним файлом іншої програми.

Нарешті, *відображувані файли* - це звичайні файли, відображені на адресу простір процесу за вказаною віртуальною адресою.

Файли належать до абстрактного механізму. Вони надають спосіб зберегти інформацію на запам'ятовуючому пристрої та зчитувати її пізніше знову. При цьому від користувача мають приховуватися такі деталі, як спосіб і місце зберігання інформації, а також деталі роботи пристрою.

Найважливішою характеристикою будь-якого механізму абстракції є іменування керованих об'єктів.

У багатьох операційних системах ім'я файлу складається з двох частин, розділених крапкою. Частина імені після крапки називається *розширенням* файлу і зазвичай означає його тип.

У деяких ОС, наприклад, Windows, розширення вказує на програму, яка створила файл.

В ієрархічно організованих файлових системах зазвичай використовують три типи імен файлів: *прості, складові та відносні.*

Просте (коротке) символьне ім'я ідентифікує файл у межах одного каталогу. Кілька файлів можуть мати одне й те саме просте ім'я, якщо вони належать різним каталогам.

Складене (повне) символьне ім'я являє собою ланцюжок, що містить ім'я та імена всіх каталогів, через які проходить шлях від кореневого каталогу до цього файлу.

Відносне ім'я файлу визначається через поточний каталог, тобто каталог, у якому в цей момент часу працює користувач.

Поняття файлу включає не тільки збережені ним дані та ім'я, а й інформацію, що описує властивості файлу. Ця інформація становить атрибути (дескриптор) файлу. Список атрибутів може бути різним у різних ОС. Приклад наведено в табл. 3.

Таблиця 3 − Приклади атрибутів файлу

Атрибут	Значення
1	2
Тип файлу	Звичайний, каталог, спеціальний тощо.
Власник файлу	Поточний власник
Власник файлу	Поточний власник
Творець файлу	Ідентифікатор користувача, який створив файл
Пароль	Пароль для отримання доступу до файлу
Час	Створення, останнього доступу, останньої зміни
Поточний розмір файлу	Кількість байт у записі
Максимальний розмір	Кількість байтів, до якої можна збільшувати розмір файлу
Прапор “тільки читання”	0 – читання / запис, 1 – тільки читання

Продовження табл. 3

Прапор “прихований”	0 – нормальний, 1 – не показувати в переліку файлів каталогу
Прапор “системний”	0 – нормальний, 1 – системний
1	2
Прапор "архівний"	0 - заархівований, 1 - потрібна архівація
Прапор "ASCII / двійковий"	0 - ASCII, 1 - двійковий
Прапор "довільного доступу"	0 - тільки послідовний доступ, 1 - довільний доступ
Прапор "тимчасовий"	0 - нормальний, 1 - видалення після закінчення роботи процесу

Користувач може отримати доступ до атрибутів, використовуючи засоби, що надаються для цієї мети файловою системою. Зазвичай дозволяється читати значення будь-яких атрибутів, а змінювати тільки деякі.

5.3 Логічна організація файлу

У загальному випадку дані, що містяться у файлі, мають деяку логічну структуру. Ця структура (організація) файлу є базою при розробці програми, призначеної для обробки цих даних. Підтримка структури даних може бути цілком покладена на додаток, або тією чи іншою мірою цю роботу може взяти на себе файлова система.

Відомо п'ять фундаментальних способів організації файлів:

- змішаний файл;
- послідовний файл;
- індексно-послідовний файл;
- індексований файл;
- файл прямого доступу.

При виборі способу організації файлу потрібно враховувати кілька критеріїв:

- швидкість доступу;
- легкість оновлення;
- економність зберігання;
- простоту обслуговування;
- надійність.

Змішаний файл. Це найменш складна форма організації файлу. Дані накопичуються в порядку надходження, коли запис складається з одного пакета даних. Записи можуть мати різні або однакові поля, розташовані в різному порядку. Кожне поле описує саме себе, включаючи як ім'я, так і значення. Довжина кожного поля має бути вказана явно або за допомогою застосування роздільника.

Оскільки змішаний файл не має жодної структури, то доступ до запису здійснюється повним перебором усіх записів файлу. Змішані файли використовуються в тому випадку, коли дані накопичуються і зберігаються перед обробкою, або і дані незручні для організації. Файли цього типу раціонально використовують кодовий простір, добре підходять для повного набору. Оновлення записів досить складне, так само як і вставка запису.

Послідовний файл. Для записів використовується фіксований формат. Усі записи мають однакову довжину (але іноді й не однакову) і складаються з однакової кількості полів фіксованої довжини, організованих у певному порядку. Оскільки довжина і позиція кожного поля відомі, то збереженню підлягають тільки значення полів. Атрибутами файлової структури є ім'я та довжина кожного поля.

Одне певне поле (або кілька полів) називається ключовим. Воно однозначно ідентифікує запис, оскільки це поле різне для кожного запису. Ба більше, записи зберігаються в "ключовій" послідовності: в алфавітному порядку для текстового ключа і в числовому - для числового. Послідовні файли часто і користуються пакетними додатками і зазвичай є оптимальним варіантом, якщо ці додатки виконують обробку всіх записів. Зручно й те, що такий файл можна зберігати як на диску, так і на магнітному диску.

Для діалогових додатків послідовний файл малоефективний, оскільки для знаходження потрібного запису потрібен послідовний перебір запису файлу.

Індексно-послідовний файл. Одним із методів подолання недоліків послідовного файлу є індексно-послідовна організація файлу. У цьому разі файл складається з трьох частин (файлів): головний файл, що містить записи з ключами, які ідуть послідовно, індексний файл, що містить індексне поле і покажчик у головний з ключами, файл переповнення.

Для пошуку потрібного запису за його ключем спочатку виконується пошук в індексному файлі. Після того як у ньому знайдено найбільше значення ключа, що не перевищує шукане, триває пошук у головному файлі.

Доповнення до файлу обробляються таким чином. У кожному записі головного файлу міститься додаткове поле, невидиме для додатка, яке є покажчиком на файл переповнення. Якщо у файлі проводиться вставка нового запису, додається у файл переповнення. Запис у головному файлі, що безпосередньо передує новому запису в логічній послідовності, оновлюється і вказує новий запис у файлі переповнення. Час від часу виконується злиття індексно-послідовного файлу з файлом переповнення.

Індексований файл. Індексно-послідовний файл зберігає одне обмеження послідовного файлу: ефективна робота з файлом обмежена роботою з ключовим полем.

Для досягнення більшої гнучкості пошуку записів необхідне використання великої кількості індексів - по одному для кожного типу поля,

яке може бути об'єктом пошуку. В узагальненому індексованому файлі доступ до записів здійснюється тільки за їхніми індексами.

Використовується два типи індексів. Повний індекс містить по одному елементу кожного типу записів головного файлу. Сам по собі індекс організовується у вигляді послідовного файлу для полегшення пошуку. Приватний індекс містить елемент для записів, у яких є поле, що цікавить користувача. Під час додавання виття запису в головний файл необхідно оновлювати всі індексні файли.

Індексовані файли використовуються тими додатками, у яких час ступання до інформації є критичною характеристикою і рідко потрібне опрацювання всіх записів у файлі.

Файл *прямого доступу.* Такий файл використовує можливість прямого доступу до блоку з відомою адресою при зберіганні файлів на диску. У кожному записі в цьому випадку також є ключове поле.

5.4 Каталожні системи

Сполучною ланкою між системою керування файлами і набором файлів слугує файловий каталог. Найпростіша форма системи каталогів полягає в тому, що є один каталог, у якому містяться всі файли. Каталог містить інформацію про файли, включно з атрибутами, місцем розташування, приналежністю. Користувачі звертаються до файлів за символьними іменами. Однак здібності людської пам'яті обмежують кількість імен об'єктів, до яких користувач може звертатися за іменами. Ієрархічна організація простору імен дає змогу значно розширити ці межі. Саме тому каталогові системи мають ієрархічну структуру. Граф, що описує ієрархію каталогів, може бути деревом або мережею. Каталоги утворюють дерево, якщо файлу дозволено входити тільки в один каталог, і мережу, якщо файл може входити в кілька каталогів (рис. 23).

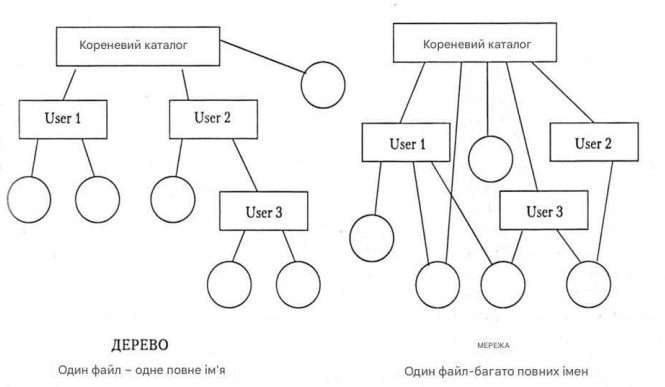

Рис. 23 − Загальний вигляд деревоподібної
та мережевої структури каталогів

Наприклад, у MS DOS і Windows каталоги утворюють деревоподібну

структуру, UNIX/Linux - мережеву.

Перше рішення полягає в тому, що на кожному з пристроїв розміщується автономна файлова система, тобто файли, які розміщені на цьому пристрої, описуються деревом каталогів, ніяк не пов'язаним із деревами каталогів на інших пристроях. У такому разі для однозначної ідентифікації файлу користувач разом зі складеним символьним ім'ям файлу має вказувати ідентифікатор логічного пристрою.

Іншим рішенням є така організація зберігання файлів, за якої користувачеві надається можливість об'єднати файлові системи, що знаходяться на різних пристроях, у єдину файлову систему, описувану єдиним деревом каталогів. Така операція називається *монтуванням.*

В ОС UNIX монтування здійснюється таким чином. Серед усіх наявних логічних дискових пристроїв виокремлюють один, який називають системним. Нехай є дві файлові системи, розташовані на різних логічних дисках, при цьому один із дисків є системним (рис. 24).

Файлова система, розташована на системному диску, називається кореневою. Для зв'язку ієрархій файлів у кореневій файловій системі вибирають деякий наявний каталог, у цьому прикладі - каталог loc. Після виконання монтування обраний каталог loc стає кореневим каталогом другої файлової системи. Через цей каталог монтована файлова система під'єднується як піддерево до загального дерева.

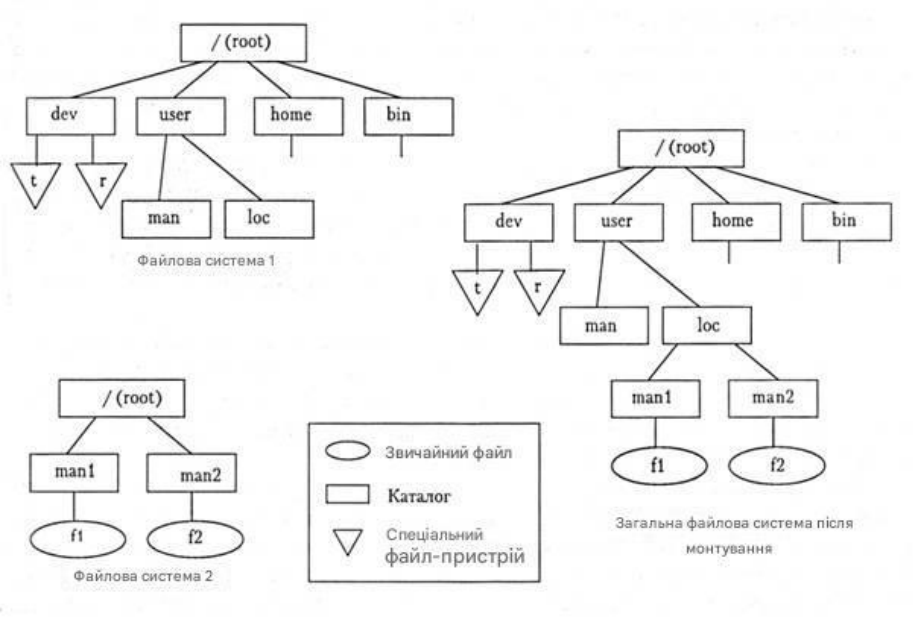

Рис. 24– Приклад монтування файлової системи

5.5 Основні можливості файлової системи NTFS

Під час проектування системи New Technology File System (NTFS) особливу увагу було приділено таким характеристикам:

- *надійність.* Високопродуктивні комп'ютери і системи спільного

користування (сервери) повинні володіти підвищеною надійністю, яка є ключовим елементом структури і поведінки NTFS. Одним із способів збільшення надійності є введення механізму транзакцій, за якого здійснюється журналювання файлових операцій (тобто відбувається фіксація в спеціальному службовому файлі змін, що відбуваються. На початку операції, пов'язаної зі зміною файлової структури, робиться відповідна позначка. Якщо під час операцій над файлами відбувається якийсь збій, то згадана позначка про початок операції залишається вказаною як незавершена. Під час виконання процедури перевірки цілісності файлової системи після перезавантаження машини ці незавершені операції буде скасовано, і файли буде приведено до початкового стану. Якщо ж операція зміни даних у файлах завершується нормальним чином, то в цьому самому службовому файлі підтримки журналювання операція відзначається як завершена);

- *розширена функціональність*. NTFS проектувалася з урахуванням можливого розширення. У ній було втілено багато додаткових можливостей - вдосконалена відмовостійкість, емуляція інших файлових систем, потужна модель безпеки, паралельне опрацювання потоків даних і створення файлових атрибутів, які визначає користувач;

- підтримка POSIX (Portable operating system for computing environments). З 1990 року POSIX є міжнародним стандартом на *машинно-незалежний інтерфейс комп'ютерного середовища.* Оскільки уряд США вимагав, щоб усі системи, які він закуповує, хоча б у мінімальному ступені відповідали стандарту POSIX, така можливість була передбачена і в NTFS. До базових засобів файлової системи POSIX належить необов'язкове використання імен файлів з урахуванням регістра, зберігання часу останнього звернення до файлу та механізм так званих "жорстких посилань" - альтернативних імен, які дають змогу посилатися на один і той самий файл за двома і більше іменами;

- *гнучкість*. Модель розподілу дискового простору в NTFS вирізняється надзвичайною гнучкістю. Розмір кластера може змінюватися від 512 байт до 64 Кбайт. NTFS також підтримує довгі імена файлів, набір символів Unicode і альтернативні імена формату 8.3 для сумісності з FAT.

NTFS чудово справляється з обробкою великих масивів даних і досить добре проявляє себе під час роботи з томами об'ємом від 300-400 Мбайт і вище. Максимально можливі розміри тома (і розміри файлу) становлять 16 Ебайт (екзабайт - 1 Ебайт=2^{64} Гбайт). Кількість файлів у кореневому і некореневому каталогах не обмежена.

Система NTFS також має певні засоби самовідновлення. NTFS підтримує різні механізми перевірки цілісності системи, включно з веденням журналів транзакцій, що дають змогу відтворити файлові операції запису за спеціальним системним журналом.

Файлова система NTFS підтримує об'єктну модель безпеки NT і розглядає всі томи, каталоги та файли як самостійні об'єкти. NTFS забезпечує безпеку на рівні файлів; це означає, що права доступу до томів, каталогів і файлів можуть залежати від облікового запису користувача і тих груп, до яких він належить. Щоразу, коли користувач звертається до об'єкта файлової системи, його права доступу перевіряються за списком дозволів цього об'єкта. Якщо користувач має достатній рівень прав, його запит задовольняється; в іншому разі запит відхиляється. Ця модель безпеки застосовується як при локальній реєстрації користувачів на комп'ютерах, так і при віддалених мережевих запитах.

Нарешті, крім величезних розмірів томів і файлів, система NTFS також має вбудовані засоби стиснення, які можна застосовувати до окремих файлів, цілих каталогів і навіть томів (і згодом скасовувати або призначати їх на свій розсуд).

5.6 Структура тому з файловою системою NTFS

Одним з основних понять, що використовуються під час роботи з NTFS, є поняття тома (volume). Можливе також створення відмовостійкого тому, що займає кілька розділів, тобто використання RAID-технології. Як і багато інших систем, NTFS ділить увесь корисний дисковий простір тому на кластери - блоки даних, адресовані як одиниці даних. NTFS підтримує розміри кластерів від 512 байт до 64 Кбайт; стандартом же вважається кластер розміром 2 або 4 Кбайт.

Весь дисковий простір у NTFS ділиться на дві нерівні частини (рис. 25). Перші 12 % диска відводяться під так звану MFT-зону - простір, який може займати, збільшуючись у розмірі, головний службовий *метафайл* MFT. Запис будь-яких даних у цю область неможливий. MFT-зона завжди тримається порожньою - це робиться для того, щоб найголовніший, службовий файл (MFT), якщо це можливо, не фрагментувався під час свого зростання. Решта 88 % тому являють собою звичайний простір для зберігання файлів.

Рис. 25 − Структура тома NTFS

MFT (master file table, загальна таблиця файлів) є централізованим каталогом всіх інших файлів диска, зокрема і себе самого. MFT поділений на записи фіксованого розміру в 1 Кбайт, і кожен запис відповідає якомусь файлу (у загальному сенсі цього слова). Перші 16 файлів мають службовий характер і недоступні операційній системі - вони називаються *метафайлами,* причому найперший метафайл - сам MFT. Ці перші 16 елементів MFT - єдина частина диска, що має строго фіксоване положення. Копія цих же 16 записів зберігається в середині тому для надійності, оскільки вони дуже важливі. Інші частини MFT-файлу можуть розташовуватися, як і будь-який інший файл, у довільних місцях диска - відновити його положення можна за допомогою його самого, "зачепившись" за найголовнішу основу - за перший елемент MFT.

Згадані перші 16 файлів NTFS (метафайли) мають службовий характер; кожен із них відповідає за якийсь аспект роботи системи. Метафайли знаходяться в кореневому каталозі NTFS-тому. Усі вони починаються із символу імені "$", хоча отримати будь-яку інформацію про них стандартними засобами складно. У табл. 4 нижче наведено основні відомі метафайли та їхнє призначення.

Таблиця 4 − Основні метафайли NTFS

Ім'я метафайла	Призначення метафайла
SMFT	Сам Master File Table
SMFTmirr	Копія перших 16 записів MFT, розміщена посередині тому
$LogFile	Файл підтримки операцій журналювання
$Volume	Службова інформація - мітка тома, версія файлової системи тощо.
SAttrDef	Список стандартних атрибутів файлів на томі
$.	Кореневий каталог
$Bitmap	Карта вільного місця тома
$Boot	Завантажувальний сектор (якщо розділ завантажувальний)
$Quota	Файл, у якому записані права користувачів на використання дискового простору
$Upcase	Файл - таблиця відповідності великих і великих літер в іменах файлів. У NTFS імена файлів записуються в Unicode (що становить 65 тисяч різних символів), і шукати великі й малі еквіваленти в цьому випадку - нетривіальне завдання.

Усі файли тому згадуються в MFT. У цій структурі зберігається вся інформація про файли, за винятком власне даних. Ім'я файлу, розмір, положення на диску окремих фрагментів тощо. - усе це зберігається у відповідному записі. Якщо для інформації не вистачає одного запису MFT,

то використовується кілька записів, причому не обов'язково тих, що йдуть підряд. Файли можуть мати не дуже великий розмір. Тоді застосовується досить вдале рішення: дані файлу зберігаються прямо в MFT, у місці, що залишилося від основних даних, у межах одного запису MFT. Файли, що займають сотні байт, зазвичай не мають свого "фізичного" втілення в основній файловій ділянці - всі дані такого файлу зберігаються в одному місці, в MFT.

Файл у томі з NTFS ідентифікується так званим файловим посиланням (File Reference), яке представляється як 64-розрядне число. Файлове посилання складається з номера файлу, який відповідає позиції його файлового запису в MFT, і номера послідовності. Останній збільшується щоразу, коли ця позиція в MFT використовується повторно, що дає змогу файловій системі NTFS виконувати внутрішні перевірки цілісності.

Кожен файл у NTFS представлений за допомогою *потоків* (streams), тобто у нього немає як таких "просто даних", а є "потоки". Для правильного розуміння потоку досить вказати, що один із потоків і має звичний нам сенс - дані файлу. Але більшість атрибутів файлу - це теж потоки. Таким чином, виходить, що базова сутність у файлу тільки одна - номер у MFT, а все інше, включаючи і його потоки, - опціонально. Цей підхід може ефективно використовуватися - наприклад, файлу можна "приліпити" ще один потік, записавши в нього будь-які дані. У Windows 2000 (і далі) у такий спосіб записано інформацію про автора і вміст файлу (одна із закладок у властивостях файлу, які можна переглянути, наприклад, із провідника). Ці додаткові потоки не видно стандартними засобами роботи з файлами: спостережуваний розмір файлу - це лише розмір основного потоку, який містить традиційні дані.

Стандартні ж атрибути для файлів і каталогів у томі NTFS мають фіксовані імена і коди типу, перелічені в табл. 5.

Таблиця 5 − Атрибути файлів у системі NTFS

Системний атрибут	Опис атрибута
Стандартна інформація про файл	Традиційні атрибути Read Only, Hidden, Archive, System, позначки часу, включно з часом створення або останньої модифікації, число каталогів, що посилаються на файл
Список атрибутів	Список атрибутів, з яких складається файл, і файлове посилання на файловий запис і MFT, у якому розташований кожен з атрибутів. Останній використовується, якщо файлу необхідно більше одного запису в MFT
Ім'я файлу	Ім'я файлу в символах Unicode. Файл може мати кілька атрибутів - імен файлу, подібно до того як це має місце в UNIX-системах.

Продовження табл. 5

Дескриптор захисту	Структура даних захисту (ACL), що оберігає файл від несанкціонованого доступу. Атрибут "дескриптор захисту" визначає, хто власник файлу і хто має доступ до нього
Дані	Власне дані файлу, його вміст. У NTFS у файлу за замовчуванням є один безіменний атрибут даних, і він може мати додаткові іменовані атрибути даних. У каталогу немає атрибута даних за замовчуванням, але він може мати необов'язкові іменовані атрибути даних
Корінь індексу, розміщення індексу	Атрибути, що використовуються для індексів імен файлів у великих каталогах
Розширені атрибути HPFS	Атрибути, що використовуються для реалізації розширених атрибутів HPFS для підсистеми OS/2 і OS/2-клієнтів файл-серверів Windows NT

Атрибути файлу в записах MFT розташовані в порядку зростання числових значень кодів типу, причому деякі типи атрибутів можуть зустрічатися в записі більш ніж один раз: наприклад, якщо у файлу є кілька атрибутів даних або кілька імен. Обов'язковими для кожного файлу в томі NTFS є атрибут стандартної інформації, атрибут імені файлу, атрибут дескриптора захисту та атрибут даних. Решта атрибутів можуть зустрічатися за потреби.

Ім'я файлу в NTFS, на відміну від файлових систем FAT і HPFS, може містити (майже) будь-які символи, включно з повним набором національних алфавітів, оскільки дані подані в Unicode - 16-бітному поданні, яке дає 65 535 різних символів. Максимальна довжина імені файлу в NTFS - 255 символів.

Великий внесок в ефективність файлової системи вносить організація каталогу. Каталог в NTFS являє собою спеціальний файл, що зберігає посилання на інші файли і каталоги, створюючи ієрархічну будову даних на диску. Файл каталогу поділено на блоки, кожен з яких містить ім'я файлу, базові атрибути і посилання на елемент MFT, який вже надає повну інформацію про елемент каталогу. Головний каталог диска - кореневий - нічим не відрізняється від звичайних каталогів, крім спеціального посилання на нього з початку метафайла MFT.

5.7 Можливості NTFS щодо обмеження доступу до файлів і каталогів

Завдяки наявності механізму розширених атрибутів, у NTFS саме за їхньою допомогою реалізовано обмеження в доступі до файлів і каталогів.

Ці додаткові атрибути, використані для обмеження в доступі до файлових об'єктів, назвали атрибутами безпеки. Під час кожного звернення до такого об'єкта порівнюється спеціальний список дискреційних прав доступу, приписаний йому, зі спеціальним системним ідентифікатором, що несе інформацію про те, від імені кого здійснюється поточний запит до файлу або каталогу. Якщо є в списку необхідний дозвіл, то дія виконується, в іншому разі система повідомляє про відмову.

Файлова система NTFS має такі дозволи, які можуть бути приписані будь-якому файлу та/або каталогу, і які називаються індивідуальними дозволами. Це дозволи **Read (прочитати), Write (записати), eXecute (виконати), Delete (видалити), Change Permissions (змінити дозволи)**, і **Take Ownership (стати власником).** Відповідні цим дозволам дії можна виконувати, тільки якщо для цього користувача або для групи (до якої він належить) є однойменний дозвіл. Інакше кажучи, якщо для деякого файлу зазначено, що всі можуть його читати і виконувати, то тільки ці дії і можна з ним зробити, якщо при цьому не вказано, що для якої-небудь іншої групи (або окремого користувача) є інші дозволи. Комбінації цих індивідуальних дозволів і визначають ті дії, які можуть бути виконані з файлом або каталогом.

Спочатку всьому диску, а отже, і файлам, які на ньому створюються, присвоєно всі індивідуальні дозволи для групи Everyone (всі). Це означає, що будь-який користувач, маючи повний набір індивідуальних дозволів на файли і каталоги, може змінювати їх на свій розсуд, тобто обмежувати інших користувачів у правах доступу до того чи іншого об'єкта. Якщо змінити дозволи на каталог, то нові файли, що створюються в ньому, отримуватимуть і відповідні дозволи: вони успадковуватимуть дозволи свого батьківського каталогу.

Є так звані стандартні дозволи, які, за задумом розробників, слід використовувати для зазначення найпоширеніших комбінацій індивідуальних дозволів.

Крім цих стандартних дозволів можна використовувати спеціальні. Вони визначаються як явна комбінація індивідуальних дозволів.

Фактичні дозволи для користувача, які він матиме на файл або каталог, визначаються як сума дозволів, які він отримує як член кількох груп. У цього загального правила є виняток. Дозвіл **No Access** (немає доступу) має пріоритет над іншими. Вона забороняє будь-який доступ до файлу або каталогу, навіть якщо користувачеві, як члену іншої групи, надано необхідний дозвіл. Можна сказати, що цей стандартний дозвіл означає не відсутність дозволів, а накладення явної заборони, і що він скасовує для користувача або групи всі дозволи, встановлені в інших рядках дискреційного списку прав доступу.

У табл. 6 наведено приклади стандартних дозволів NTFS.

Таблиця 6 − Стандартні дозволи NTFS

Стандартні дозволи NTFS	Відповідні їм комбінації індивідуальних дозволів NTFS	
	Для каталогів	Для файлів
No Access (немає доступу)	Немає ніяких дозволів	Немає жодних дозволів
Список (перегляд)	Read, eXecute	Немає жодних дозволів
Read (читання)	Read, eXecute	Read, eXecute
Add (додавання)	Write, eXecute	Немає жодних дозволів
Add & Read (читання і додавання)	Read, Write, eXecute	Read, eXecute
Change (зміна)	Read, Write, eXecute, Delete	Read, Write, eXecute, Видалити
Full Control (повний доступ)	Усі дозволи	Усі дозволи

Каталоги зазвичай володіють тими самими дозволами, що й файли і папки, які знаходяться в них, хоча у кожного файлу можуть бути свої дозволи. Дозволи, які є у файлу, мають пріоритет над дозволами, які встановлені на каталог, в якому знаходиться цей файл. Наприклад, якщо ви створюєте каталог усередині іншого каталогу, для якого адміністратори мають право повного доступу, а користувачі - право читання, то новий каталог успадкує ці права. Те саме стосується і файлів, що копіюються з іншого каталогу або переміщуються з іншого розділу NTFS.

Якщо каталог або файл переміщується в інший каталог того ж розділу NTFS, то атрибути безпеки не успадковуються від нового каталогу. Річ у тім, що під час переміщення файлів у межах одного розділу NTFS змінюється тільки покажчик місцезнаходження об'єкта, а всі інші атрибути (включно з атрибутами безпеки) залишаються без змін.

Три наступні важливі правила допоможуть визначити стан прав доступу під час переміщення або копіювання об'єктів NTFS:

- під час переміщення файлів у межах розділу NTFS зберігаються вихідні права доступу.
- під час виконання інших операцій (створення або копіювання файлів, а також їх переміщення між розділами NTFS) успадковуються права доступу батьківського каталогу.
- під час переміщення файлів із розділу NTFS у розділ FAT усі права NTFS втрачаються.

Контрольні питання

1. Які основні функції виконує файлова система в операційній системі?

2. Які переваги надає використання файлової системи при роботі з даними?

3. Чому важливо, щоб кілька процесів мали одночасний доступ до файлів? Як файлова система забезпечує цю функцію?

4. Як файлова система забезпечує захист файлів від несанкціонованого доступу?

5. Які основні типи файлів підтримують файлові системи?

6. Що таке каталог і яку роль він відіграє у файловій системі?

7. Які є основні типи імен файлів у файлових системах?

8. Яка інформація зберігається в атрибутах файлу? Які атрибути можуть змінюватися користувачем?

9. Які критерії потрібно враховувати при виборі способу організації файлів?

10. Яким чином файл прямого доступу забезпечує швидкий доступ до даних?

11. У чому полягає відмінність між індексно-послідовним файлом та індексованим файлом?

12. Як організовано ієрархічну структуру каталогів у сучасних операційних системах?

6 УПРАВЛІННЯ ВВЕДЕННЯМ-ВИВЕДЕННЯМ

Однією з головних функцій ОС є керування всіма пристроями введення-виведення комп'ютера. ОС повинна передавати пристроям команди, перехоплювати переривання і обробляти помилки; вона також повинна забезпечувати інтерфейс між пристроями та іншою частиною системи. З метою розвитку інтерфейс має бути однаковим для всіх типів пристроїв (незалежність від пристроїв).

6.1 Фізична організація пристроїв введення-виведення

Пристрої введення-виведення діляться на два типи:

- ***блок-орієнтовані***, тобто пристрої зберігають інформацію в блоках фіксованого розміру, кожен з яких має свою власну адресу (наприклад, диск);
- ***байт-орієнтовані***, тобто пристрої не адресовані і не дають змоги здійснювати операцію пошуку, вони генерують або споживають послідовність байтів (наприклад, термінали, рядкові принтери, мережеві адаптери).

Однак деякі зовнішні пристрої не належать до жодного класу, наприклад, **годинник**, який, з одного боку, не адресований, а з іншого боку,

не породжує потоку байтів. Цей пристрій лише видає сигнал переривання в деякі моменти часу.

Зовнішній пристрій зазвичай складається з таких частин:

- механічного компонента (власне пристрою);
- електронного компонента (контролера пристрою або адаптера).

Деякі контролери можуть керувати кількома пристроями. Якщо інтерфейс між контролером і пристроєм стандартизований, то незалежні виробники можуть випускати сумісні як контролери, так і пристрої.

ОС зазвичай має справу з контролером, а не з пристроєм.

Контролер, як правило, виконує прості **функції**, наприклад, такі:

- перетворює потік біт на блоки, що складаються з байт
- здійснює контроль;
- виправляє помилки.

Кожен контролер має кілька 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 введення-виведення. В інших комп'ютерах адреси регістрів введення-виведення, які часто називають портами, утворюють власний адресний простір завдяки введенню спеціальних операцій введення-виведення (наприклад, команд IN і OUT у процесорах i86).

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. Під час завершення команди контролер організовує переривання для того, щоб передати управління процесором операційній системі, яка повинна перевірити результати операції. Процесор отримує результати і статус пристрою, читаючи інформацію з регістрів контролера.

6.2 Організація програмного забезпечення введення-виведення

Основна ідея організації ПЗ введення-виведення полягає в **розбитті його на кілька рівнів**, до того ж нижні рівні забезпечують екранування особливостей апаратури від верхніх, а ті, зі свого боку, забезпечують зручний інтерфейс для користувачів.

Ключовим принципом є **незалежність від пристроїв**. Вигляд програми не повинен залежати від того, читає вона дані з гнучкого диска чи з жорсткого диска.

Дуже близькою 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**. Взагалі кажучи, помилки слід обробляти

якомога ближче до апаратури. Якщо контролер виявляє помилку читання, то він має спробувати її скоригувати. Якщо ж це йому не вдається, то виправленням помилок має зайнятися драйвер пристрою. Багато помилок можуть зникати під час повторних спроб виконання операцій введення-виведення, наприклад, помилки, спричинені наявністю пилинок на голівках читання або на диску. І тільки якщо нижній рівень не може впоратися з помилкою, він повідомляє про помилку верхньому рівню.

Ще одне ключове питання - це **використання блокуючих (синхронних) і неблокуючих (асинхронних) передач**. Більшість операцій фізичного введення-виведення виконується асинхронно - процесор починає передачу і переходить на іншу роботу, поки не настає переривання. Користувацькі програми набагато легше писати, якщо операції введення-виведення блокуючі - після команди READ програма автоматично призупиняється доти, доки дані не потраплять у буфер програми. ОС виконує операції введення-виведення асинхронно, але представляє їх для користувацьких програм у синхронній формі.

Остання проблема полягає в тому, що **одні пристрої є поділюваними, а інші - виділеними**. Диски - це пристрої, що розділяються, тому що одночасний доступ кількох користувачів до диска не є проблемою. Принтери - це виділені пристрої, тому що не можна змішувати рядки, що друкуються різними користувачами. Наявність виділених пристроїв створює для операційної системи деякі проблеми.

Для розв'язання поставлених проблем доцільно розділити програмне забезпечення введення-виведення на чотири шари (рис. 26):

- обробка переривань
- драйвери пристроїв;
- незалежний від пристроїв шар ОС;
- користувацький шар ПЗ.

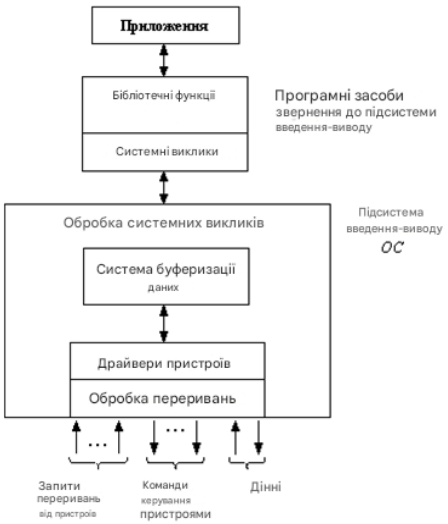

Рис. 26 − Багаторівнева організація підсистеми введення-виведення

6.3 Обробка переривань

Переривання мають бути приховані якомога глибше в надрах ОС, щоб якомога 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. У разі настання переривання процедура обробки переривання виконує розблокування процесу, який ініціював операцію вводу-виводу, використовуючи виклики UP, SIGNAL або надсилаючи процесу повідомлення. У будь-якому разі ефект від переривання полягатиме в тому, що раніше заблокований процес тепер продовжить своє виконання.

6.4 Драйвери пристроїв

Увесь залежний від пристрою код поміщається в драйвер пристрою. Кожен драйвер керує пристроями одного типу або, можливо, одного класу.

В ОС тільки драйвер пристрою знає про конкретні особливості будь-якого пристрою. Наприклад, тільки драйвер диска має справу з доріжками, секторами, циліндрами, часом встановлення голівки та іншими факторами, що забезпечують правильну роботу диска.

Драйвер пристрою приймає запит від пристроїв 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, що знову надійшов, приєднується до черги вже наявних запитів, і він буде виконаний, коли настане його черга.

Перший крок у 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, голівок, секторів, перевірку, чи працює мотор, чи знаходиться головка над потрібним циліндром. Коротше кажучи, він повинен вирішити, які операції контролера потрібно виконати і в якій послідовності.

Після передавання команди контролеру драйвер має вирішити, чи блокувати себе до закінчення заданої операції, чи ні. Якщо операція займає значний час, як під час друку деякого блока даних, то драйвер блокується доти, доки операція не завершиться, і обробник переривання не розблокує його. Якщо команда введення-виведення виконується швидко (наприклад, прокрутка екрана), то драйвер очікує її завершення без блокування.

6.5 Незалежний від пристроїв шар ОС

Більша частина програмного забезпечення введення-виведення є незалежною від пристроїв. Точна межа між драйверами та 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 такі:

- 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 кількох різних блоків у 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.

Під час створення файлу або заповнення 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 розроблено алгоритм пошуку вільного блоку, незалежний від пристрою і реалізований програмним шаром, що знаходиться вище шару драйверів.

6.6 Користувацький шар програмного забезпечення

Хоча більша частина ПЗ введення-виведення знаходиться всередині ОС, деяка його частина міститься в **бібліотеках**, які пов'язуються з призначеними для користувача програмами. Системні виклики, що включають виклики введення-виведення, зазвичай робляться бібліотечними процедурами. Якщо програма, написана мовою С, 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 слугувати функція printf мови С, 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 велику кількість процедур, які виконують введення-виведення і працюють як частина користувацької програми.

Іншою категорією програмного забезпечення введення-виведення є

підсистема спулінгу (spooling). Спулінг - це спосіб роботи з виділеними пристроями в мультипрограмній системі. Розглянемо типовий пристрій, що потребує спулінгу - рядковий принтер. Хоча технічно легко дозволити кожному користувацькому процесу відкрити спеціальний файл, пов'язаний із принтером, такий спосіб небезпечний через те, що користувацький процес може монополізувати принтер на довільний час. Замість цього створюється спеціальний процес - монітор, який отримує виключні права на використання цього пристрою. Також створюється спеціальний каталог, який називається каталогом спулінгу. Для того, щоб надрукувати файл, користувацький процес поміщає виведену інформацію в цей файл і поміщає його в каталог спулінгу. Процес-монітор по черзі роздруковує всі файли, що містяться в каталозі спулінгу.

Контрольні питання

1. Що є однією з головних функцій ОС стосовно пристроїв введення-виведення?
2. Які типи пристроїв введення-виведення розрізняють? Наведіть приклади.
3. Які компоненти входять до складу зовнішнього пристрою?
4. Які функції виконує контролер пристрою?
5. Що таке регістри контролера і як вони взаємодіють із центральним процесором?
6. Як ОС взаємодіє з контролером при виконанні операцій введення-виведення?
7. Що таке переривання і яку роль вони відіграють у процесі введення-виведення?
8. Чим відрізняються блок-орієнтовані та байт-орієнтовані пристрої?
9. Що таке драйвер пристрою і які функції він виконує?
10. Як розділяється програмне забезпечення введення-виведення на рівні?
11. Яку роль відіграє незалежний від пристроїв шар ОС?
12. Як ОС обробляє помилки під час операцій введення-виведення?
13. Що таке блокуючі та неблокуючі передачі даних?
14. Що таке спулінг і для яких пристроїв він використовується?
15. Як ОС розподіляє та звільняє виділені пристрої?

7 ПОБУДОВА ОПЕРАЦІЙНИХ СИСТЕМ

7.1 Принципи побудови операційних систем

Принцип модульності. Під *модулем* у загальному випадку розуміють функціонально закінчений елемент системи, виконаний відповідно до

прийнятих міжмодульних інтерфейсів. За своїм визначенням модуль передбачає можливість легко замінити його на інший за наявності заданих інтерфейсів. Способи відокремлення складових частин ОС в окремі модулі можуть істотно відрізнятися, але найчастіше поділ відбувається саме за функціональною ознакою. Значною мірою поділ системи на модулі визначається використовуваним методом проектування ОС (від низу до верху або навпаки).

Особливо важливе значення під час побудови ОС мають *привілейовані* (ті, що працюють у привілейованому режимі в разі відключеної системи переривань), *повторно вхідні* (дають змогу багаторазового паралельного використання) і *реентерабельні* (дають змогу багаторазово переривати своє виконання) модулі, тому що вони дають змогу більш ефективно використовувати ресурси обчислювальної системи.

Принцип модульності відображає технологічні та експлуатаційні властивості системи. Найбільшого ефекту від його використання можна досягти в разі, коли принцип поширено одночасно на операційну систему, прикладні В ОС виділяється деяка частина важливих модулів, які мають постійно перебувати в оперативній пам'яті для ефективнішої організації обчислювального процесу. Цю частину в ОС називають ядром, оскільки це дійсно основа системи. Під час формування складу ядра потрібно враховувати дві суперечливі вимоги:

- до складу ядра мають увійти найчастіше використовувані системні модулі;
- кількість модулів має бути такою, щоб об'єм пам'яті, який займає ядро, був би не надто великим.

До складу ядра, як правило, входять модулі з управління системою переривань, засоби з переведення програм зі стану рахунку в стан очікування, готовності і назад, засоби з розподілу таких основних ресурсів, як оперативна пам'ять і процесор. Крім програмних модулів, що входять до складу ядра і постійно розташовуються в оперативній пам'яті, може бути багато інших системних програмних модулів, які отримують назву *транзитних.* Транзитні програмні модулі завантажуються в оперативну пам'ять тільки за потреби і в разі відсутності вільного простору можуть бути заміщені іншими транзитними модулями.

Принцип генерування ОС. Основне положення цього принципу визначає такий спосіб вихідного представлення центральної системної керуючої програми ОС (її ядра і основних компонентів, які повинні постійно перебувати в оперативній пам'яті), який давав би змогу налаштовувати цю системну супервізорну частину, виходячи з конкретної конфігурації конкретного обчислювального комплексу та кола розв'язуваних завдань. Цю процедуру проводять рідко, перед досить тривалим періодом експлуатації ОС. Процес генерації здійснюється за допомогою спеціальної програми-

генератора і відповідної вхідної мови для цієї програми, що дає змогу описувати програмні можливості системи і конфігурацію машини. У результаті генерації виходить повна версія ОС. Згенерована версія ОС являє собою сукупність системних наборів модулів і даних.

Згаданий раніше принцип модульності позитивно проявляється при генерації ОС. Він істотно спрощує налаштування ОС на необхідну конфігурацію обчислювальної системи. У наші дні під час використання персональних комп'ютерів із принципом генерування ОС можна зіткнутися хіба що тільки під час роботи з Linux. У цій UNIX-системі є можливість не тільки використовувати будь-яке готове ядро ОС, а й самому згенерувати (скомпілювати) таке ядро, яке буде оптимальним для цього конкретного персонального комп'ютера і розв'язуваних на ньому завдань. Крім генерації ядра в Linux є можливість вказати і набір драйверів і служб, які довантажуються, тобто частину функцій можна реалізовувати модулями, які безпосередньо входять до ядра системи, а частину - модулями, що мають статус довантажуваних, транзитних.

В інших сучасних поширених ОС для персональних комп'ютерів конфігурація ОС під відповідний склад устаткування здійснюється на етапі інсталяції, а потім склад драйверів і зміну деяких параметрів ОС може бути здійснено за допомогою редагування конфігураційного файлу.

Цей принцип враховує можливість проведення однієї і тієї ж роботи різними засобами. До складу ОС може входити кілька типів моніторів (модулів супервізора, які керують тим чи іншим видом ресурсу), різні засоби організації комунікацій між обчислювальними процесами. Наявність кількох типів моніторів, кількох систем керування файлами дає змогу користувачам швидко і найадекватніше адаптувати ОС до певної конфігурації обчислювальної системи, забезпечити максимально ефективне завантаження технічних засобів під час розв'язування конкретного класу завдань, отримати максимальну продуктивність під час розв'язування заданого класу завдань.

Побудова віртуальних ресурсів, їх розподіл і використання тепер використовується практично в будь-якій ОС. Цей принцип дає змогу уявити структуру системи у вигляді певного набору планувальників процесів і розподільників ресурсів (моніторів) і використовувати єдину централізовану схему розподілу ресурсів.

Найбільш природним і закінченим проявом концепції віртуальності є поняття *віртуальної* машини. Будь-яка операційна система, будучи засобом розподілу ресурсів і організовуючи за певними правилами управління процесами, приховує від користувача і його додатків реальні апаратні та інші ресурси, замінюючи їх деякою абстракцією. У результаті користувачі бачать і використовують віртуальну машину як якийсь пристрій, здатний сприймати їхні програми, написані певною мовою програмування,

виконувати їх і видавати результати.

Найчастіше віртуальна машина, що надається користувачеві, відтворює архітектуру реальної машини, але архітектурні елементи в такому поданні виступають з новими або поліпшеними характеристиками, які часто спрощують роботу з системою. Характеристики можуть бути довільними, але найчастіше користувачі бажають мати власну "ідеальну" за архітектурними характеристиками машину в такому складі:

- однакова за логікою роботи пам'ять (віртуальна) практично необмеженого обсягу. Середній час доступу можна порівняти зі значенням цього параметра оперативної пам'яті. Організація роботи з інформацією в такій пам'яті проводиться в термінах обробки даних - в термінах роботи з сегментами даних на рівні обраної користувачем мови програмування;

- довільна кількість процесів (віртуальних), здатних працювати паралельно і взаємодіяти під час роботи. Способи керування процесами, зокрема синхронізація та інформаційні взаємодії, реалізовано та доступно користувачам на рівні використовуваної мови в термінах керування процесами;

- довільна кількість зовнішніх пристроїв (віртуальних), здатних працювати з пам'яттю віртуальної машини паралельно або послідовно, асинхронно або синхронно стосовно роботи того чи іншого віртуального процесора, які ініціюють роботу цих пристроїв. Інформація, що передається або зберігається на віртуальних пристроях, не обмежена допустимими розмірами. Доступ до такої інформації здійснюється на основі або послідовного, або прямого способу доступу в термінах відповідної системи управління файлами. Передбачено розширення інформаційних структур даних, що зберігаються на віртуальних пристроях.

Ступінь наближення до "ідеальної" віртуальної машини може бути більшим або меншим у кожному конкретному випадку. Що більше віртуальна машина, реалізована засобами ОС на базі конкретної апаратури, наближена до "ідеальної" за характеристиками машини і, отже, що більше її архітектурно-логічні характеристики відрізняються від тих, що реально існують, то більший ступінь віртуальності в отриманої користувачем машини.

Одним з аспектів віртуалізації є організація можливості виконання в даній ОС додатків, які розроблялися для інших ОС. Прикладом реалізації принципу віртуалізації може слугувати *VDM-машина* (virtual DOS machine) - захищена підсистема, що надає повне середовище MS-DOS і консоль для виконання MS-DOS додатків. Одночасно може виконуватися практично довільна кількість VDM-сесій. Такі VDM-машини є і в системах Microsoft Windows, і в OS/2.

Цей принцип полягає в тому, що зв'язок програм з конкретними пристроями здійснюється не на рівні трансляції програми, а в період

планування її виконання. У результаті перекомпіляція під час роботи програми з новим пристроєм, на якому розташовуються дані, не потрібна.

Принцип дає змогу однаково здійснювати операції керування зовнішніми пристроями незалежно від їхніх конкретних фізичних характеристик. Наприклад, програмі, що містить операції обробки послідовного набору даних, байдуже, на якому носії ці дані будуть розташовуватися. Зміна носія і даних, що розміщуються на них (за незмінності структурних характеристик даних), не принесе жодних змін у програму, якщо в системі реалізовано принцип незалежності.

Одним з аспектів сумісності є здатність ОС виконувати програми, написані для інших ОС або для більш ранніх версій цієї операційної системи, а також для іншої апаратної платформи.

Необхідно розділяти питання двійкової сумісності та сумісності на рівні вихідних текстів додатків. Двійкова сумісність досягається в тому випадку, коли можна взяти виконувану програму і запустити її на виконання на іншій ОС. Для цього необхідні: 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 вихідних текстів вимагає наявності відповідного транслятора у складі системного програмного забезпечення, а також сумісності на рівні бібліотек і системних викликів. При цьому необхідна перекомпіляція наявних вихідних текстів у новий виконуваний модуль.

Набагато складніше досягти двійкової сумісності між процесорами, заснованими на різних архітектурах. Для того щоб один комп'ютер виконував програми іншого (наприклад, програму для ПК типу IBM PC бажано виконати на ПК типу Macintosh фірми Apple), цей комп'ютер повинен працювати з машинними командами, які йому від початку незрозумілі. Виходом у таких випадках є використання так званих прикладних середовищ або емуляторів. З огляду на те, що основну частину програми, як правило, складають виклики бібліотечних функцій, прикладне середовище імітує бібліотечні функції цілком, використовуючи заздалегідь написану бібліотеку функцій аналогічного призначення, а решту команд емулює кожну окремо.

Одним із засобів забезпечення сумісності програмних і користувацьких інтерфейсів є відповідність стандартам POSIX. Використання стандарту POSIX дає змогу створювати програми в стилі UNIX, які згодом можуть легко переноситися з однієї системи в іншу.

Відкрита ОС доступна для аналізу як користувачам, так і системним фахівцям, які обслуговують обчислювальну систему. ОС, що нарощується (модифікується, розвивається), дає змогу не тільки використовувати можливості генерації, а й вводити до її складу нові модулі, удосконалювати наявні тощо. Необхідно, щоб можна було легко внести доповнення і зміни,

якщо це буде потрібно, і не порушити цілісність системи. Прекрасні можливості для розширення надає підхід до структурування ОС за типом клієнт-сервер з використанням мікроядерної технології. Відповідно до цього підходу ОС будується як сукупність привілейованої керуючої програми і набору непривілейованих послуг - "серверів". Основна частина ОС залишається незмінною і водночас можуть бути додані нові сервери або поліпшені старі.

Цей принцип іноді трактують як розширюваність системи. До відкритих ОС насамперед слід віднести UNIX-системи і, природно, ОС Linux.

Операційна система відносно легко повинна переноситися з процесора одного типу на процесор іншого типу і з апаратної платформи (яка включає поряд з типом процесора і спосіб організації всієї апаратури комп'ютера, інакше кажучи, архітектуру обчислювальної системи) одного типу на апаратну платформу іншого типу.

Написання переносної ОС аналогічно написанню будь-якого коду, який можна переносити, - потрібно дотримуватися деяких правил. По-перше, більша частина ОС має бути написана мовою, наявною на всіх системах, на які планується надалі її переносити. Це, насамперед, означає, що ОС має бути написана мовою високого рівня, переважно стандартизованою, наприклад, мовою С. Програма, написана на асемблері, не є в загальному випадку такою, що переноситься. По-друге, важливо мінімізувати або, якщо можливо, виключити ті частини коду, які безпосередньо взаємодіють з апаратними засобами. Залежність від апаратури може мати багато форм. Деякі очевидні форми залежності включають пряме маніпулювання регістрами та іншими апаратними засобами. Нарешті, якщо апаратно-залежний код не може бути повністю виключений, то він має бути ізольований у кількох добре локалізованих модулях. Апаратно-залежний код не повинен бути розподілений по всій системі. Наприклад, можна заховати апаратно-залежну структуру в програмно задані дані абстрактного типу. Інші модулі системи працюватимуть із цими даними, а не з апаратурою, використовуючи набір деяких функцій. Коли ОС переноситься, то змінюються тільки ці дані та функції, які ними маніпулюють.

Введення стандартів POSIX мало на меті забезпечити переносимість створюваного програмного забезпечення.

Забезпечення безпеки під час виконання обчислень є бажаною властивістю для будь-якої багатокористувацької системи. Правила безпеки визначають такі властивості, як захист ресурсів одного користувача від інших і встановлення квот за ресурсами для запобігання захопленню одним користувачем усіх системних ресурсів (таких, як пам'ять).

Забезпечення захисту інформації від несанкціонованого доступу є

обов'язковою функцією мережевих операційних систем. У багатьох сучасних ОС гарантується ступінь безпеки даних, що відповідає рівню С2 у системі стандартів США.

Основи стандартів у сфері безпеки були закладені в документі "Критерії визначення безпеки комп'ютерних систем" (Trusted Computer System Evaluation Criteria). Цей документ, виданий Національним центром комп'ютерної безпеки (NCSC - National Computer Security Center) у США 1983 року, часто називають Помаранчевою книгою. Аналогом Помаранчевої книги є міжнародний стандарт ISO/IEC 15408, опублікований 2005 року.

Згідно з вимогами Помаранчевої книги безпечною вважається система, яка "за допомогою спеціальних механізмів захисту контролює доступ до інформації таким чином, що тільки особи, які мають відповідні повноваження, або процеси, які виконуються від їхнього імені, можуть отримати доступ на читання, запис, створення або видалення інформації".

Ієрархія рівнів безпеки, наведена в Помаранчевій книзі, позначає найнижчий рівень безпеки як D, а вищий - як А.

До класу D потрапляють системи, оцінка яких виявила їхню невідповідність вимогам усіх інших класів.

Основними властивостями, характерними для систем класу С, є наявність підсистеми обліку подій, пов'язаних із безпекою, і вибірковий контроль доступу. Клас (рівень) С ділиться на 2 підрівні: рівень С1, що забезпечує захист даних від помилок користувачів, але не від дій зловмисників, і більш суворий рівень С2. На рівні С2 мають бути присутніми:

- засоби секретного входу, що забезпечують ідентифікацію користувачів шляхом введення унікального імені та пароля перед тим, як їм буде дозволено доступ до системи;

- вибірковий контроль доступу, що дає змогу власнику 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), що 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, що полягає в тому, що пам'ять ініціалізується перед тим, як повторно використовується.

На цьому рівні система не захищена від помилок користувача, але поведінка його може бути проконтрольована за записами в журналі, залишеними засобами спостереження та аудиту.

Системи рівня В засновані на помічених даних і розподілі користувачів за категоріями, тобто реалізують мандатний контроль доступу. Кожному користувачеві присвоюється рейтинг захисту, і він може

отримувати доступ до даних тільки відповідно до цього рейтингу. Цей рівень на відміну від рівня С захищає систему від помилкової поведінки користувача.

Рівень А є найвищим рівнем безпеки, він вимагає на додаток до всіх вимог рівня В виконання формального, математично обґрунтованого доказу відповідності системи вимогам безпеки.

7.2 Побудова інтерфейсів операційних систем

ОС завжди виступає як інтерфейс між апаратурою комп'ютера і користувачем з його завданнями. Під інтерфейсами операційних систем розуміють спеціальні інтерфейси системного і прикладного програмування, призначені для виконання таких завдань:

- управління процесами, яке містить у собі такий набір основних функцій:

а) запуск, призупинення і зняття завдання з виконання;

б) завдання або зміна пріоритету завдання;

в) взаємодія завдань між собою (механізми сигналів, семафори, черги, конвеєри, поштові скриньки);

г) RPC (remote procedure call) - віддалений виклик підпрограм;

- управління пам'яттю:

а) запит на виділення блоку пам'яті;

б) звільнення пам'яті;

в) зміна параметрів блоку пам'яті (наприклад, пам'ять може бути заблокована процесом або надана в загальний доступ);

г) відображення файлів на пам'ять (є не у всіх системах);

- управління введенням/виведенням:

а) запит на управління віртуальними пристроями (нагадаємо, що управління введенням/виводом є привілейованою функцією самої ОС, і жодна з призначених для користувача завдань не повинна мати можливості безпосередньо керувати пристроями);

б) файлові операції (запити до системи управління файлами на створення, зміну і видалення даних, організованих у файли).

Користувацький інтерфейс операційної системи реалізується за допомогою спеціальних програмних модулів, які приймають його команди на відповідній мові (або з використанням графічного інтерфейсу) і транслюють їх у звичайні виклики відповідно до основного інтерфейсу системи. Зазвичай ці модулі називають інтерпретатором команд.

В даний час в основному використовуються графічні інтерфейси (GUI), в яких задіяні відповідні маніпулятори типу "миша" або "трекбол". Вказівка курсором на об'єкти і клацання (клік) або подвійне клацання за відповідними клавішами призводить до якихось дій - запуску програми, асоційованої з об'єктом, на який вказували, вибору та/або активізації пунктів

меню тощо. Можна сказати, що така інтерфейсна підсистема транслює "команди" користувача у звернення до ОС.

Пояснимо також, що управління GUI - окремий випадок задачі управління введенням/виводом, що не є частиною ядра операційної системи, хоча в низці випадків розробники ОС відносять функції GUI до основного системного API.

Звернення до операційної системи, відповідно до наявного API, може здійснюватися як за допомогою виклику підпрограми з передачею їй необхідних параметрів, так і через механізм програмних переривань. Вибір методу реалізації викликів функцій API має визначатися архітектурою платформи.

Так, наприклад, в операційній системі MS-DOS, яка розроблялася для однозадачного режиму, використовувався механізм програмних переривань. При цьому основний набір функцій API був доступний через точку входу обробника int 21h.

У складніших системах є не одна точка входу, а безліч - за кількістю функцій API. Так, у більшості операційних систем використовується метод виклику підпрограм. У цьому випадку виклик спочатку передається в модуль API (наприклад, це бібліотека часу виконання - run time library, тобто RTL - містить у собі не тільки модулі із системи програмування, а й модулі ОС), який і перенаправляє виклик відповідним обробникам програмних переривань, що входять до складу операційної системи.

7.3 Інтерфейс прикладного програмування

Загальний термін API (application program interface, інтерфейс прикладного програмування) поділяють на такі напрями:

- API як інтерфейс високого рівня, що належить до бібліотек RTL;
- API прикладних і системних програм, що входять до постачання операційної системи;
- інші API.

Інтерфейс прикладного програмування, як це і випливає з назви, призначений для використання прикладними програмами системних ресурсів ОС і реалізованих нею функцій. API описує сукупність функцій і процедур, що належать ядру або надбудовам ОС.

Отже, API являє собою набір функцій, що надаються системою програмування розробнику прикладної програми та орієнтовані на організацію взаємодії результуючої прикладної програми *з цільовою обчислювальною системою.*

Цільова обчислювальна система являє собою сукупність програмних і апаратних засобів, в оточенні яких виконується результуюча програма. Сама результуюча програма породжується системою програмування на підставі коду вихідної програми, створеного розробником, а також об'єктних модулів

і бібліотек, що входять до складу системи програмування.

Функції API дають змогу розробнику будувати результуючу прикладну програму так, щоб використовувати засоби цільової обчислювальної системи для виконання типових операцій. При цьому розробник програми позбавлений необхідності створювати вихідний код для виконання цих операцій.

Програмний інтерфейс API містить у собі не тільки самі функції, а й угоди про їхнє використання, що регламентуються операційною системою (ОС), архітектурою цільової обчислювальної системи та системою програмування.

Існує кілька варіантів реалізації API:

- реалізація на рівні ОС;
- реалізація на рівні системи програмування;
- реалізація на рівні зовнішньої бібліотеки процедур і функцій.

Система програмування в кожному з цих варіантів надає розробнику засоби для підключення функцій API до вихідного коду програми та організації їхніх викликів. Об'єктний код функцій API підключається до результуючої програми компонувальником за необхідності.

Можливості API можна оцінювати з таких позицій:

- ефективність виконання функцій API - охоплює швидкість виконання функцій і обсяг обчислювальних ресурсів, потрібних для їх виконання;
- широта можливостей, що надаються;
- залежність прикладної програми від архітектури цільової обчислювальної системи.

В ідеалі хотілося б мати набір функцій API, що виконуються з найвищою ефективністю, надають користувачеві всі можливості сучасних ОС і мають мінімальну залежність від архітектури обчислювальної системи (ще краще - позбавлені такої залежності).

Реалізація функцій API на рівні ОС. Під час реалізації функцій API на рівні ОС за їх виконання відповідальність несе ОС. Об'єктний код, що виконує функції, або безпосередньо входить до складу ОС (або навіть ядра ОС), або постачається у складі бібліотек, що динамічно завантажуються, розроблених для цієї ОС. Система програмування відповідальна тільки за те, щоб організувати інтерфейс для виклику цього коду.

У такому варіанті результуюча програма звертається безпосередньо до ОС. Тому досягається найбільша ефективність виконання функцій API порівняно з усіма іншими варіантами реалізації API.

Недоліком організації API за такою схемою є практично повна відсутність переносимості не тільки коду результуючої програми, а й коду вихідної програми. Програма, створена для однієї архітектури обчислювальної системи, не зможе виконуватися на обчислювальній системі

іншої архітектури навіть після того, як її об'єктний код буде повністю перебудований. Найчастіше система програмування не зможе виконати перебудову вихідного коду для нової архітектури обчислювальної системи, оскільки багато функцій API, орієнтованих на певну ОС, будуть у новій архітектурі просто відсутні.

Таким чином, у цій схемі для перенесення прикладної програми з однієї цільової обчислювальної системи на іншу буде потрібна зміна вихідного коду програми.

Перенесення можна було б досягти, якщо уніфікувати функції API в різних ОС. З урахуванням корпоративних інтересів виробників ОС такий напрямок їхнього розвитку видається практично неможливим. У найкращому разі за докладання певних організаційних зусиль вдається домогтися стандартизації смислового (семантичного) наповнення основних функцій API, але не їхнього програмного інтерфейсу.

Добре відомим прикладом API такого роду може слугувати набір функцій, що надаються користувачеві з боку ОС типу Microsoft Windows - WinAPI (Windows API). Треба сказати, що навіть усередині цього корпоративного API існує певна неузгодженість, яка дещо обмежує переносимість програм між різними ОС типу Windows.

Ще одним прикладом такого API можна вважати набір сервісних функцій ОС типу MS-DOS, реалізований у вигляді набору підпрограм обслуговування програмних переривань.

Реалізація функцій API на рівні системи програмування. Якщо функції API реалізуються на рівні системи програмування, вони надаються користувачеві у вигляді бібліотеки функцій відповідної мови програмування. Зазвичай ідеться про бібліотеку часу виконання - RTL (run time library).

Система програмування надає користувачеві бібліотеку відповідної мови програмування і забезпечує підключення до результуючої програми об'єктного коду, відповідального за виконання цих функцій.

Очевидно, що ефективність функцій API у такому варіанті буде дещо нижчою, ніж при безпосередньому зверненні до функцій ОС. Так відбувається, оскільки для виконання багатьох функцій API бібліотека RTL мови програмування повинна все одно виконувати звернення до функцій ОС. Наявність усіх необхідних викликів і звернень до функцій ОС в об'єктному коді RTL забезпечує система програмування.

Однак переносимість вихідного коду програми в такому варіанті буде найвищою, оскільки синтаксис і семантика всіх функцій будуть строго регламентовані в стандарті відповідної мови програмування. Вони залежать від мови і не залежать від архітектури цільової обчислювальної системи. Тому для виконання прикладної програми на новій архітектурі обчислювальної системи достатньо заново побудувати код результуючої

програми за допомогою відповідної системи програмування.

Однакове виконання функцій мови забезпечується системою програмування. Під час орієнтації на різні архітектури цільової обчислювальної системи в системі програмування можуть знадобитися різні комбінації викликів функцій ОС для виконання одних і тих самих функцій вихідної мови. Це має бути враховано в коді RTL. У загальному випадку для кожної архітектури цільової обчислювальної системи буде потрібен свій код RTL мови програмування. Вибір того чи іншого об'єктного коду RTL для підключення до результуючої програми система програмування забезпечує автоматично.

Проблема, головним чином, полягає в тому, що більшість мов програмування надають користувачеві не дуже широкий набір стандартизованих функцій. Тому розробник вихідного коду суттєво обмежений у виборі доступних функцій API.

Як правило, набору стандартних функцій виявляється недостатньо для створення повноцінної прикладної програми. Тоді розробник може звернутися до функцій інших бібліотек, наявних у складі системи програмування. У цьому разі немає гарантії, що функції, які включені до складу даної системи програмування, але не входять до стандарту мови програмування, будуть доступні в іншій системі програмування. Особливо якщо вона орієнтована на іншу архітектуру цільової обчислювальної системи. Така ситуація вже ближча до третього варіанта реалізації API.

Реалізація функцій API за допомогою зовнішніх бібліотек. Під час реалізації функцій API за допомогою зовнішніх бібліотек вони надаються користувачеві у вигляді бібліотеки процедур і функцій, створеної стороннім розробником. Причому розробником такої бібліотеки може виступати той самий виробник.

Система програмування відповідальна тільки за те, щоб підключити об'єктний код бібліотеки до результуючої програми. Причому зовнішня бібліотека може бути і динамічно завантажуваною (завантажуваною під час виконання програми).

З погляду ефективності виконання цей метод реалізації API має найнижчі результати, оскільки зовнішня бібліотека звертається як до функцій ОС, так і до функцій RTL мови програмування. Тільки за дуже високої якості зовнішньої бібліотеки її ефективність стає порівнянною з бібліотекою RTL.

Якщо говорити про переносимість вихідного коду, то тут вимога тільки одна - зовнішня бібліотека, що використовується, має бути доступна в будь-якій з архітектур обчислювальних систем, на які орієнтована прикладна програма. Тоді вдається досягти переносимості. Це можливо, якщо використовувана бібліотека задовольняє якомусь прийнятому стандарту, а система програмування підтримує цей стандарт.

Наприклад, бібліотеки, що задовольняють стандарту POSIX, доступні в більшості систем програмування для мови С. І якщо прикладна програма використовує тільки бібліотеки цього стандарту, то її вихідний код буде переносимим.

Для більшості специфічних бібліотек окремих розробників це не так. Якщо користувач використовує якусь бібліотеку, то вона орієнтована на обмежений набір доступних архітектур цільової обчислювальної системи. Прикладами можуть слугувати бібліотеки MFC (Microsoft foundation classes) фірми Microsoft і VCL (Visual Components Library) фірми Borland, жорстко орієнтовані на архітектуру ОС типу Windows.

Проте, багато фірм-розробників зараз прагнуть створити бібліотеки, які б забезпечували переносимість вихідного коду додатків між різними архітектурами цільової обчислювальної системи. До них можна віднести бібліотеку CLX (Component Library for Cross Platform) виробництва фірми Borland, орієнтовану на архітектуру ОС типу Linux і ОС типу Windows.

Загалом розвиток функцій прикладного API йде в напрямку спроби створити бібліотеки API, що забезпечують широку переносимість вихідного коду. Однак, з огляду на корпоративні інтереси різних виробників і ситуацію, що склалася на ринку системного програмного забезпечення, найближчим часом навряд чи вдасться досягти значних успіхів у цьому напрямку.

Тому розробники системних програм змушені залишатися в досить вузьких рамках обмежень стандартних бібліотек мов програмування.

Що стосується прикладних програм, то набагато більшу перспективу для них надають технології, пов'язані з розробками в рамках архітектури "клієнт - сервер" або трирівневої архітектури створення додатків. У цьому напрямі провідні виробники ОС, СУБД і систем програмування швидше досягнуть угод, ніж у напрямі стандартизації API.

Отже, було розглянуто основні принципи, цілі та підходи до реалізації системних API. Слід зазначити ще один дуже важливий момент: бажано, щоб інтерфейс прикладного програмування не залежав від системи програмування. Зазвичай базові функції API не залежать від системи програмування і можуть використовуватися з будь-якої системи програмування, хоча і з застосуванням відповідних правил побудови викликаючих послідовностей. Водночас у низці випадків система програмування може сама генерувати звернення до функцій API.

Як правило, API не стандартизовані. У кожному конкретному випадку набір викликів API визначається насамперед архітектурою ОС та її призначенням. Водночас приймаються спроби стандартизувати деякий базовий набір функцій, оскільки це істотно полегшує перенесення додатків з однієї ОС в іншу.

Таким прикладом може слугувати дуже відомий і, мабуть, один із

найпоширеніших стандартів - стандарт POSIX. У цьому стандарті перераховано великий набір функцій, їхніх параметрів і значень, що повертаються.

Стандартизованими, згідно з POSIX, є не тільки звернення до API, а й файлова система, організація доступу до зовнішніх пристроїв, набір системних команд. Використання в додатках цього стандарту дає змогу надалі легко переносити такі програми з однієї ОС в іншу шляхом найпростішої перекомпіляції вихідного тексту.

Окремим випадком спроби стандартизації API є внутрішній корпоративний стандарт компанії Microsoft, відомий як WinAPI. Він містить такі реалізації: Win 16, Win32s, Win32, WinCE.

З точки зору WinAPI (через низку ідеологічних причин - обов'язковий графічний "віконний" інтерфейс користувача), базовим завданням є вікно. Таким чином, WinAPI спочатку орієнтований на роботу в графічному середовищі. Однак базові поняття доповнено традиційними функціями, зокрема частково підтримується стандарт POSIX.

7.4 Класифікація системних викликів

У всіх операційних системах, виходячи з видів ресурсів, з якими вони пов'язані, функції інтерфейсу прикладного програмування можна розділити на кілька груп. Так, у Windows можна виділити такі групи функцій API:

- управління процесами, потоками і синхронізацією;
- управління віртуальною і фізичною пам'яттю;
- управління введенням-виведенням.

В API Windows також є ще низка груп:

- графічний інтерфейс користувача;
- управління захистом;
- управління реєстром;
- консольні функції
- інші групи спеціальних функцій.

Приклади функцій API, що належать до різних груп, наведено в табл. 7-11 .

Таблиця 7 − Основні функції управління процесами і потоками

Функція	Опис
Створити процес	Створити новий процес
CreateThread	Створити новий потік в існуючому процесі
ExitProcess	Завершити поточний процес і всі його потоки
ExitThread	Завершити цей потік
SetPriorityClass	Задати клас пріоритету для процесу
SetThreadPriority	Задати пріоритет для потоку

Таблиця 8 − Основні функції управління синхронізацією

Функція	Опис
CreateSemaphore	Створити новий семафор
OpenSemaphore	Відкрити існуючий семафор
ReleaseSemaphorc	Збільшити на одиницю лічильник семафора
EnterCriticalSection	Захопити блокування для критичної секції
LeaveCriticalSection	Звільнити блокування для критичної секції

Таблиця 9 − Основні функції керування віртуальною та фізичною пам'яттю

Функція	Опис
VirtualAlloc	Зарезервувати або фіксувати область у ВП
VirtualFree	Звільнити область або скасувати фіксацію області
VirtualProtect	Змінити режим доступу до області ВП
VirtualQuery	Дізнатися стан області ВП
VirtualLock	Зафіксувати сторінки ВП
VirtualUnlock	Розфіксувати сторінки ВП
HeapCreate	Створити пул у пам'яті
HeapAllocate	Виділити блок пам'яті в пулі
HeapReAllocate	Змінити розмір блоку пам'яті в пулі
HeapFree	Звільнити блок пам'яті в пулі
GetProcessHeap	Отримати дескриптор пулу

Таблиця 10 − Основні функції для файлового введення-виведення

Функція	Опис
CreateFile	Створити або відкрити файл; повернути дескриптор файлу
DeleteFile	Видалити наявний файл
ReadFile	Прочитати дані з файлу
WriteFile	Записати дані у файл
SetFilePointer	Встановити покажчик у файлі в певну позицію
GetFileSize	Визначити розмір файлу
SetEndOfFile	Змінити розмір файлу за поточним значенням покажчика
GetFileAttributes	Повернути атрибути файлу
CopyFile	Копіювати файл
LockFile	Заблокувати область файлу для забезпечення взаємного виключення
UnlockFile	Скасувати блокування області файлу

Таблиця 11 − Основні функції управління каталогами

Функція	Опис
CreateDirectory	Створити новий каталог
RemoveDirectory	Видалити порожній каталог
FindFirstFile	Ініціалізація, щоб почати читання каталогу
FindNextFile	Прочитати наступний запис каталогу
MoveFile	Перемістити файл з одного каталогу в інший
SetCurrent Directory	Змінити поточний робочий каталог

Налічується кілька сотень функцій графічного інтерфейсу користувача (GUI). Тому слід обмежитися назвами груп цих функцій (табл. 12)

Таблиця 12 − Групи API-функцій, що реалізують графічний інтерфейс користувача

Група API	Опис
Керування вікнами	Створення, знищення вікон, керування вікнами
Меню	Створення, знищення та додавання пунктів меню
Діалогові вікна	Відображення діалогових вікон і збір інформації
Малювання та креслення	Відображення точок, ліній і геометричних фігур
Текст	Виведення тексту з використанням певного шрифту, розміру та кольору
Растрові зображення, курсори та значки	Відображення на екрані растрових зображень, курсорів і значків
Кольори та палітри	Керування набором доступних кольорів
Буфер обміну	Передавання інформації від однієї програми до іншої
Введення	Отримання інформації від миші та клавіатури

Таблиця 13 − Основні функції управління захистом

Функція	Опис
InitializeSecurity Descriptor	Підготувати новий дескриптор захисту
LookupAccountSid	Знайти SID за заданим ім'ям користувача
SetSecurityDescriptorOwner	Ввести SID власника в дескриптор захисту
SetSecurityDescriptorGroup	Ввести SID групи в дескриптор захисту

Таблиця 14 − Основні функції управління реєстром

Функція	Опис
RegCreateKeyEx	Створити новий ключ реєстру
RegDeleteKey	Видалити ключ реєстру
RegOpenKeyEx	Відкрити ключ і отримати його дескриптор
RegEnumKeyEx	Перенумерувати підключення, підпорядковані ключу дескриптора
RegQueryValueEx	Шукати дані за значенням у ключі

Таблиця 15 − Основні функції управління консоллю

Функція	Опис
SetConsoleTitle	Визначити текст заголовка для консолі
WriteConsole	Вивести текст у консоль
ReadConsole	Введення з консолі
ReadConsoleInput	Обробка подій миші та клавіатури

7.5 Інтерфейс користувача

Інтерфейс користувача - це сукупність засобів, що надається розробником програми для взаємодії користувача з цією програмою. Існують додатки, які під час свого виконання не взаємодіють із користувачем, але такі випадки трапляються рідко.

Вивчення програми, по суті справи, зводиться до вивчення її інтерфейсу. Інтерфейс програм формується з тих елементів, які надають система програмування та операційна система. Наприклад, MS DOS працює в текстовому режимі і не підтримує мишу, тому основним типом інтерфейсу є так званий командний рядок. Розробникам ПЗ MS DOS не надавали графічні засоби для побудови меню, елементів керування тощо, тому програмісти змушені були самі вести розробку подібних елементів інтерфейсу, що призводить до великих накладних витрат під час програмування та до появи величезної кількості різномастих елементів інтерфейсу, які не завжди однаково функціонують. У сучасних операційних системах елементи інтерфейсу підтримуються самими системами.

Оскільки з операційною системою можуть взаємодіяти різні категорії користувачів - адміністратори, системні програмісти, оператори, програмісти та кінцеві користувачі, то кожній із цих категорій надається свій інтерфейс.

Вид інтерфейсу взаємодії користувача з ОС історично визначався можливостями технічних засобів спілкування користувача з системою.

Спочатку користувач спілкувався з системою, вводячи в командний рядок консолі інформацію за принципом "запрошення/запит - введення - обробка - відповідь". З розвитком операційних систем було розроблено графічні інтерфейси, що являють собою надбудови над операційними

системами.

У Windows така надбудова, звана також графічним середовищем, є невід'ємною частиною ОС. Вона реалізується за допомогою драйвера відеосистеми, функцій інтерфейсу графічних пристроїв (GDI) і графічного інтерфейсу користувача (GUI).

У UNIX-подібних системах (UNIX , Linux) графічний інтерфейс будується на основі стандарту X Window System. Система X Window побудована на моделі "клієнт/сервер". Основою підсистеми є так званий X-сервер, який працює на комп'ютері користувача і забезпечує виведення зображення на екран монітора, а також введення даних від користувача (через клавіатуру, мишу тощо). Додатки виступають у ролі клієнтів. Для спілкування із сервером клієнтам надається бібліотека XLib, що містить функції низькорівневої взаємодії.

Наступний рівень у реалізації інтерфейсних можливостей в UNIX-подібних системах представлений *інтегрованими графічними середовищами.* До складу цих середовищ крім службових програм входить безліч прикладних програм для вирішення різних завдань. Інтегровані графічні середовища розширюють і доповнюють перераховані вище компоненти системи X Window. Так, вихідна бібліотека Xlib є досить бідною, і інтегровані графічні середовища додають до неї свою бібліотеку графічних функцій. Існує великий список інтегрованих графічних середовищ різних виробників.

7.6 Користувацький інтерфейс додатків

У кожному Windows-додатку реалізація графічного призначеного для користувача інтерфейсу становить більшу частину його програмного коду. Під час роботи з додатками звертає на себе увагу *стандартизація інтерфейсу.* По-перше, усі вікна мають рамку, яка визначає їхні межі; рамка використовується також для зміни розмірів вікна. У верхньому лівому кутку вікна знаходиться значок системного меню. Клацання миші на цей значок відкриває системне меню. Праворуч від значка системного меню розташований заголовок вікна. У правому верхньому кутку вікна розташовані кнопки мінімізації, повноекранної розгортки (повноекранного представлення) і закриття вікна. Нижче розташована робоча область - це частина вікна, в якій відображається виконання програми. Стандартними елементами призначеного для користувача інтерфейсу стали головне меню програми, контекстні меню, гарячі клавіші, панелі інструментів, рядок стану, вертикальні та горизонтальні лінійки прокрутки, різні елементи керування тощо.

Сучасні візуальні середовища програмування пропонують десятки (або сотні) елементів керування. Оскільки ці елементи трапляються майже у всіх Windows-додатках, то знайомство з цими елементами, їхнім

призначенням і способами керування є важливим завданням. До найчастіше використовуваних стандартних і загальних елементів керування належать кнопки, прапорці, перемикачі, вікна введення, списки, смуги прокрутки, вікна відкриття файлів, інструменти, закладки (сторінки), підказки та інші.

Слід зазначити, що майже всі елементи керування є повноцінними вікнами. Крім головного вікна та елементів керування Windows підтримує й інші види вікон: модальні та немодальні діалогові вікна, вікна повідомлень, діалогові вікна загального користування тощо. Познайомившись з будь-яким новим елементом керування, користувач може бути впевнений, що в іншому застосунку цей елемент матиме те саме призначення. У цьому і полягає основний сенс стандартизації користувацького інтерфейсу.

Поява елементів керування стала можливою завдяки реалізації нової архітектури взаємодії користувача з програмою. У багатьох операційних системах (MS DOS, UNIX) взаємодію з користувачем *ініціює програма* користувача

Спочатку інтерфейс користувача в консольних додатках мав структуру "запрошення або опитування - очікування введення - введення - обробка - виведення результату". За такої архітектури взаємодії *програма керує користувачем.* Раніше ця архітектура була цілком прийнятною, поки не з'явилося безліч джерел інформації, відмінних від клавіатури. Наприклад, тепер завдяки наявності миші користувач на свій розсуд може вибрати потрібну команду меню або натиснути будь-яку кнопку, клацнути на значку панелі управління, ввести текст або, врешті-решт, натиснути допустиму комбінацію клавіш. Для реалізації такого сценарію консольні додатки не годяться. Потрібна інша архітектура, в якій *користувач керує програмою,* а остання завжди готова обробляти введення з будь-яких джерел інформації. Саме таку взаємодію і реалізовано в Windows.

Прикладна програма не запитує за допомогою системних викликів введення даних з клавіатури, натомість вона чекає, коли Windows введе інформацію і передасть її програмі. Цей процес складається з кількох етапів. Програма очікує повідомлення від Windows і в загальному випадку не діє. Користувач, клацаючи клавішами миші або вводячи інформацію з клавіатури, ініціює подію і роботу відповідного драйвера. Повідомлення від драйверів спочатку передаються в системну чергу, а потім перерозподіляються по чергах додатків (точніше, потоків). З черги додатка операційна система вибирає повідомлення і разом з керуванням передає його відповідному додатку. Після отримання повідомлення програма здійснює його обробку і далі знову переходить у стан очікування чергового повідомлення. Така взаємодія програми та операційної системи, що базується на повідомленнях, більш ніж будь-що інше, визначає схему побудови всіх програм для Windows.

Описана вище і реалізована в Windows схема взаємодії користувача,

операційної системи і додатка називається *архітектурою, керованою подіями*.

Контрольні питання

1. Що є предметом вивчення в курсі Операційні системи?
2. Назвіть фактори, які істотно вплинули на архітектуру сучасних ОС.
3. Як називались програми, які частково замінили функції оператора?
4. Назвіть типові команді програми-монітора.
5. Назвіть час найбільшого поширення програм-моніторів.
6. У чому полягає принцип пакетної обробки даних?
7. Як називалась мова керування завданнями?
8. Що таке мультипрограмний режим роботи ОС?
9. Назвіть найпоширенішу мультипрограмну пакетну ОС, для яких комп'ютерів вона була призначена?
10. Назвіть переваги та недоліки ОС з розподілом часу.
11. Назвіть найпоширеніші ЕОМ, на яких могла виконуватися ОС розподілу часу.
12. Які можуть бути ОС за кількістю користувачів?
13. Які можуть бути ОС залежно від апаратної основи?
14. Які можуть бути ОС за призначенням?
15. Що таке ОС загального призначення?
16. Що таке ОС реального часу?
17. Що таке пакетні ОС?
18. Що таке діалогові ОС?
19. Що таке вбудовані ОС?

8 UNIX ПОДІБНІ СИСТЕМИ

UNIX є прикладом винятково вдалої реалізації простої багатозадачної та багатокористувацької операційної системи. Свого часу її проектували як інструментальну систему для розробки програмного забезпечення.

Своєю унікальністю система UNIX зобов'язана багато в чому тій обставині, що її, по суті, створили лише два розробники (Кен Томпсон і Денніс Рітчі), причому люди, які створювали її, робили систему для себе, і перший час її використовували на міні-ЕОМ з дуже скромними обчислювальними ресурсами. З цієї причини UNIX, перш за все, має просту, але дуже потужну командну мову і незалежну від пристроїв файлову систему. Оскільки при створенні цієї ОС використовували мову високого

рівня, якою пишуть не тільки системні, а й прикладні програми (йдеться про мову С), то система і додатки, що виконуються в ній, вийшли легко переносними (мобільними). Компілятор з мови С для всіх відтрансльованих програм дає реентерабельний і поділюваний код, що дає змогу ефективно використовувати наявні в системі ресурси.

Першою метою при розробці цієї системи було прагнення зберегти простоту і обійтися мінімальною кількістю функцій. Усі реальні складнощі залишали користувацьким програмам.

Другою метою була спільність. Одні й ті самі методи та механізми мали використовуватися в багатьох випадках. Тому спільність в UNIX-системах проявляється в багатьох аспектах, і зокрема:

- звернення до файлів, пристроїв введення/виводу і буферів міжпроцесних повідомлень виконуються за допомогою схожих методів;
- одні й ті самі механізми іменування, привласнення альтернативних імен і захисту від несанкціонованого доступу застосовуються до файлів із даними і директоріями, і пристроїв:
- одні й ті самі механізми працюють щодо програмно й апаратно ініційованих переривань.

Нарешті, третя мета полягала у створенні операційного середовища, в якому великі завдання можна було б розв'язувати, комбінуючи наявні невеликі програми, а не розробляючи програми заново.

8.1 Основні поняття системи UNIX

8.1.1 Віртуальна машина

Система UNIX - багатокористувацька. Кожному користувачеві після реєстрації (входу в систему) надають віртуальний комп'ютер, у якому є всі необхідні ресурси: процесор (процесорний час виділяють із використанням динамічних пріоритетів для того, щоб забезпечити рівність в обслуговуванні), пам'ять, пристрої, файли. Поточний стан такого віртуального комп'ютера, що надається користувачеві, називається образом. Можна сказати, що процес - це виконання образу. Образ складається з:

- образу пам'яті;
- значень загальних регістрів процесора;
- стану відкритих файлів;
- поточної директорії (каталогу файлів) та іншої інформації.

Образ процесу під час його виконання розміщується в основній пам'яті.

Образ пам'яті ділиться на три логічні сегменти:

- сегмент реентерабельних процедур (починається з нульової адреси у віртуальному адресному просторі процесу);
- сегмент даних (розташовується слідом за сегментом процедур і може зростати в бік великих адрес);

- сегмент стека (починається зі старшої адреси і зростає в бік молодших адрес у міру занесення в нього інформації під час викликів підпрограм і при перериваннях).

8.1.2 Користувач

Людина, яка зареєстрована в облікових файлах системи і, отже, має обліковий запис (account), називається зареєстрованим користувачем системи. Реєстрацію нових користувачів зазвичай виконує адміністратор системи. Користувач не може змінити своє облікове ім'я, але може встановити та/або змінити свій пароль. Паролі зберігаються в окремому файлі в закодованому вигляді.

Усі користувачі ОС UNIX явно чи неявно працюють із файлами. Файлова система ОС UNIX має деревоподібну структуру. Проміжними вузлами дерева є каталоги з посиланнями на інші каталоги або файли, а листя дерева відповідають файлам або порожнім каталогам. Кожному зареєстрованому користувачеві відповідає деякий каталог файлової системи, який називається "домашнім" (home) каталогом користувача. Під час входу в систему користувач отримує необмежений доступ до свого домашнього каталогу та всіх каталогів і файлів, що містяться в ньому. Користувач може створювати, видаляти і модифікувати каталоги і файли, що містяться в домашньому каталозі. Потенційно можливий доступ і до всіх інших файлів, однак він може бути обмежений, якщо користувач не має достатніх привілеїв.

8.1.3 Інтерфейс користувача

Традиційний спосіб взаємодії користувача із системою UNIX ґрунтується на використанні командних мов (оскільки нині поширеними є графічні інтерфейси, то і в ОС UNIX почали дедалі частіше працювати в X Window). Після входу користувача в систему для нього запускається один із командних інтерпретаторів. Зазвичай у системі підтримують кілька командних інтерпретаторів зі схожими, але відмінними за своїми можливостями командними мовами. Загальна назва для будь-якого командного інтерпретатора ОС UNIX - shell (оболонка), оскільки будь-який інтерпретатор представляє зовнішнє оточення ядра системи.

Викликаний командний інтерпретатор видає запрошення на введення користувачем командного рядка, який може містити просту команду, конвеєр команд або послідовність команд. Після виконання чергового командного рядка і видачі на екран термінала або у файл відповідних результатів, shell знову видає запрошення на введення командного рядка, і так доти, доки користувач не завершить свій сеанс роботи і не вийде із системи.

Командні мови, що використовуються в ОС UNIX, досить прості, щоб нові користувачі могли швидко почати працювати, і досить потужні, щоб можна було використовувати їх для написання складних програм. Остання

можливість спирається на механізм командних файлів (shell scripts), які можуть містити довільні послідовності командних рядків. При зазначенні імені командного файлу замість чергової команди інтерпретатор читає файл рядок за рядком і послідовно інтерпретує команди.

8.1.4 Привілейований користувач

Ядро ОС UNIX ідентифікує кожного користувача за його ідентифікатором (UID - user identifier), унікальним цілим значенням, що присвоюється користувачеві під час реєстрації в системі. Крім того, кожен користувач належить до деякої групи користувачів, яка також ідентифікується деяким цілим значенням (GID - group identifier). Значення UID і GID для кожного зареєстрованого користувача зберігаються в облікових файлах системи і приписуються процесу, в якому виконується командний інтерпретатор, запущений при вході користувача в систему. Ці значення успадковуються кожним новим процесом, запущеним від імені цього користувача, і використовуються ядром системи для контролю правомочності доступу до файлів, виконання програм тощо.

Очевидно, що адміністратор системи, який теж є зареєстрованим користувачем, повинен мати більші можливості, ніж звичайні користувачі. В ОС UNIX це завдання вирішується шляхом виділення єдиного нульового значення UID. Користувач із таким UID називається суперкористувачем (superuser) або root. Він має необмежені права на доступ до будь-якого файлу і на виконання будь-якої програми. Крім того, такий користувач має можливість повного контролю над системою. Він може зупинити її і навіть зруйнувати.

Ще однією важливою відмінністю суперкористувача від звичайного користувача ОС UNIX є те, що на суперкористувача не поширюються обмеження на використовувані ресурси. Для звичайних користувачів встановлюються такі обмеження, як максимальний розмір файлу, максимальна кількість сегментів пам'яті, що розділяється, максимально допустимий простір на диску тощо. Суперкористувач може змінювати ці обмеження для інших користувачів, але на нього вони не діють.

8.1.5 Команди

Командний рядок складається з імені команди (тобто імені виконуваного файлу), за яким йде список аргументів, розділених пробілами. Оболонка розбиває командний рядок на компоненти. Зазначений у команді файл завантажується, і йому забезпечується доступ до заданих у команді аргументів.

Будь-яка командна мова сімейства shell фактично складається з трьох частин:

- службових конструкцій, що дають змогу маніпулювати з текстовими рядками і будувати складні команди на основі простих команд;
- вбудованих команд, які виконуються безпосередньо інтерпретатором

командної мови;

- команд, що представляються окремими виконуваними файлами.

8.1.6 Процеси

Процес в ОС UNIX розуміють у класичному розумінні цього терміна, тобто як програму, що виконується у власному віртуальному адресному просторі. Коли користувач входить у систему, автоматично створюється процес, у якому виконується програма командного інтерпретатора. Якщо командному інтерпретатору зустрічається команда, що відповідає виконуваному файлу, то він створює новий процес і запускає в ньому відповідну програму, починаючи з функції main. Ця запущена програма, своєю чергою, може створити процес і запустити в ньому іншу програму (вона теж має містити функцію main) тощо.

Кожен процес має одного батька, але може породити багато процесів. Початковий (нульовий) процес є особливим процесом, який створюється в результаті завантаження системи. Після породження нового процесу з ідентифікатором 1 нульовий процес стає процесом підкачки та реалізує механізм віртуальної пам'яті. Процес з ідентифікатором 1, відомий під ім'ям init, є пращуром будь-якого іншого процесу в системі і пов'язаний з кожним процесом особливим чином.

8.1.7 Виконання процесів

Процес може виконуватися в одному з двох станів, а саме *користувацькому* та *системному*. У призначеному для користувача стані процес виконує призначену для користувача програму і має доступ до призначеного для користувача сегмента даних. У системному стані процес виконує програми ядра і має доступ до системного сегмента даних.

Коли користувацькому процесу потрібно виконати системну функцію, він створює системний виклик. Фактично відбувається виклик ядра системи як підпрограми. З моменту появи системного виклику процес вважається системним. Таким чином, користувацький і системний процеси є двома фазами одного і того ж процесу, але вони ніколи не перетинаються між собою. Кожна фаза користується своїм власним стеком. Стек завдання містить аргументи, локальні змінні та іншу інформацію щодо функції, що виконуються в режимі завдання. Диспетчерський процес не має користувацької фази.

В UNIX-системах використовується поділ часу, тобто кожному процесу виділяється квант часу. Або процес завершується сам до закінчення відведеного йому кванта часу, або він відкладається після закінчення кванта. Механізм диспетчеризації характеризується досить справедливим розподілом процесорного часу між усіма процесами. Користувацьким процесам приписуються пріоритети залежно від кількості одержуваного ними процесорного часу. Процесам, які отримали велику кількість процесорного часу, призначають нижчі пріоритети, у той час як процесам,

які отримали лише невелику кількість процесорного часу, навпаки, підвищують пріоритет. Такий метод диспетчеризації забезпечує хороший час реакції для всіх користувачів системи. Усі системні процеси мають вищі пріоритети порівняно з призначеними для користувача і тому завжди обслуговуються першочергово.

8.1.8 Структура файлової системи

Файл у системі UNIX являє собою безліч символів із довільним доступом. У файлі містяться довільні дані, поміщені туди користувачем, і файл не має жодної іншої структури, крім тієї, яку накладає на нього користувач.

Далі розглядається одна з перших реалізацій файлової системи, оскільки основні її ідеї зберігаються досі.

Інформація на дисках розміщується поблочно, по 512 байт у кожному блоці. Блок спеціально взято рівним розміру сектора. Диск розбивається на такі області:

- невикористаний блок;
- керуючий блок або суперблок, у якому міститься розмір диска і межі інших областей;
- і-список, що складається з описів файлів, які називаються і-вузлами;
- область для зберігання вмісту файлів.

Кожен і-вузол містить:

- ідентифікацію власника;
- ідентифікацію групи власника;
- біти захисту;
- фізичні адреси на диску, де знаходиться вміст файлу;
- розмір файлу;
- час створення файлу;
- час останнього використання файлу (modification time);
- час останньої зміни атрибутів (change time);
- число зв'язків-посилань, що вказують на файл;
- індикацію, чи є файл директорією, звичайним файлом або спеціальним файлом.

Слідом за і-списком ідуть блоки, призначені для зберігання вмісту файлів. Простір на диску, що залишився вільним від файлів, утворює пов'язаний список вільних блоків.

Таким чином, файлова система UNIX є структурою даних, розміщеною на диску, яка містить керівний суперблок, у якому визначено файлову систему загалом; масив і-вузлів, де визначено всі файли у файловій системі; самі файли; сукупність вільних блоків. Виділення простору під дані здійснюється блоками фіксованого розміру.

Файл-директорія, у якому перелічено імена файлів, дає змогу встановити відповідність між іменами та самими файлами. Директорії

утворюють деревоподібну структуру. На кожен звичайний файл або файл пристрою можуть бути посилання в різних вузлах цієї структури. У непривілейованих програмах запис у директорії не дозволено, але за наявності відповідних дозволів вони можуть читати їх. Додаткових зв'язків між директоріями немає.

Велику кількість системних директорій система UNIX використовує для своїх власних потреб. Одна з них, коренева директорія, є базою для всієї структури директорій, і, "відштовхуючись" від неї, можна знайти розміщення всіх файлів. В інших системних директоріях містяться програми і команди, що надаються користувачам, і файли пристроїв.

8.2 Операційна система Linux

Linux - це вільно розповсюджувана версія UNIX, яка спочатку була розроблена Лінусом Торвальдсом (Linus Torvalds) в університеті Гельсінкі (Фінляндія) в 1991 році. Усі компоненти системи, включно з вихідними текстами, поширюються з ліцензією на вільне копіювання та встановлення для необмеженої кількості користувачів, яку розробив Річард Столман, засновник Фонду вільного розповсюдження програм Free Software Foundation.

Linux було створено за допомогою багатьох UNIX-програмістів та ентузіастів з Інтернету. До цього проєкту добровільно долучилися ті, хто має достатньо навичок і здібностей розвивати систему.

Наразі існує понад 50 комерційних компаній, які самі розробляють і продають свої пакети Linux, що дещо розмиває межу між комерційним і безкоштовним програмним забезпеченням.

Linux - це повноцінна багатозадачна багатокористувацька операційна система (так само, як і всі інші версії UNIX). Це означає, що одночасно багато користувачів можуть працювати на одній машині, одночасно виконуючи багато програм. Linux досить добре сумісний з низкою стандартів для UNIX (наскільки можна говорити про стандартизацію UNIX) на рівні вихідних текстів. Усі вихідні тексти для Linux, включно з ядром, драйверами пристроїв, бібліотеками, призначеними для користувача програмами та інструментальними засобами поширюються вільно.

Linux підтримує різні типи файлових систем для зберігання даних. Деякі файлові системи, такі як файлова система *ext2fs,* були створені спеціально для Linux.

Виконувані програми використовують динамічно зв'язувані бібліотеки, тобто виконувані програми можуть спільно використовувати бібліотечну програму, представлену одним фізичним файлом на диску. Це дає змогу виконуваним файлам займати менше місця на диску, особливо тим, які багаторазово використовують бібліотечні функції. Є також статичні зв'язувані бібліотеки для тих, хто бажає користуватися налагодженням на

рівні об'єктних кодів або мати "повні" виконувані програми, що не потребують бібліотек, що розділяються. У Linux бібліотеки, що розділяються, динамічно зв'язуються під час виконання, даючи змогу програмісту замінювати бібліотечні модулі своїми власними.

Контрольні питання

1. У чому принципові відмінності між звичайним користувачем і суперкористувачем в UNIX-подібних системах?
2. Чому можливість заміни бібліотечних модулів є важливою для програмістів у Linux?
3. Які команди роботи з каталогами в UNIX-подібних системах вам відомі?
4. Хто були основними розробниками операційної системи UNIX, і яка була їхня головна мета?
5. Чому UNIX вважається простою, але потужною системою?
6. Яким чином UNIX забезпечує мобільність і переносимість програм?
7. Які основні принципи закладені в архітектуру UNIX?
8. Чому в UNIX важливо використовувати комбінацію невеликих програм замість створення нових великих додатків?
9. Що таке реентерабельний і поділюваний код, і яку роль він відіграє в UNIX?
10. Хто розробив операційну систему Linux, і коли вона була створена?
11. Яка ліцензія використовується для поширення Linux, і хто її створив?
12. Чим Linux відрізняється від комерційних версій UNIX?
13. Які основні характеристики Linux як багатозадачної та багатокористувацької системи?
14. Які типи файлових систем підтримує Linux?
15. Що таке динамічно зв'язувані бібліотеки, і яку користь вони дають у Linux?

9 ТРАНСЛЯТОРИ

9.1 Основні принципи побудови трансляторів, компіляторів та інтерпретаторів

9.1.1 Визначення транслятора

Транслятор - це програма, яка переводить програму вихідною (вхідною) мовою в еквівалентну їй програму результуючою (вихідною) мовою. У цьому визначенні слово "програма" зустрічається тричі, і це не помилка і не тавтологія. У роботі транслятора дійсно беруть участь три програми.

1. Сам транслятор є програмою, тобто транслятор - це частина програмного забезпечення (ПЗ), він є набором машинних команд і даних, його виконує комп'ютер, як і всі інші програми в межах операційної системи (ОС). Усі складові частини транслятора являють собою динамічно завантажувані бібліотеки або модулі цієї програми зі своїми вхідними і вихідними даними.

2 Вихідними даними для роботи транслятора слугує програма вихідною мовою програмування - деяка послідовність речень вхідної мови. Ця програма називається *вхідною,* або *вихідною, програмою.* Зазвичай це символьний файл, але цей файл повинен містити текст програми, що задовольняє синтаксичним і семантичним вимогам вхідної мови. Крім того, цей файл несе в собі деякий сенс, який визначається семантикою вхідної мови. Часто файл, що містить текст вихідної програми, називають вихідним файлом.

3. Вихідними даними транслятора є програма на результуючій мові. Ця програма називається ***результуючою програмою***. Результуюча програма будується за синтаксичними правилами вихідної мови транслятора, а її зміст визначається семантикою вихідної мови.

Важливим пунктом у визначенні транслятора є еквівалентність вихідної та результуючої програм. ***Еквівалентність*** цих двох програм означає збіг їхнього сенсу з погляду семантики вхідної мови (для вихідної програми) і семантики вихідної мови (для результуючої програми). Без виконання цієї вимоги сам транслятор втрачає будь-який практичний сенс.

Отже, щоб створити транслятор, необхідно насамперед обрати вхідну та вихідну мови. З точки зору перетворення речень вхідної мови на еквівалентні їм речення вихідної мови транслятор виступає як перекладач.

Наприклад, трансляція програми з мови С на мову асемблера по суті нічим не відрізняється від перекладу, скажімо, з російської мови на англійську, з тією лише різницею, що складність мов дещо інша. Тому й саме слово "транслятор" (англ.: translator) означає "перекладач".

Результатом роботи транслятора буде ***результуюча програма***, але тільки в тому разі, якщо текст вихідної програми є правильним - не містить помилок з погляду синтаксису та семантики вхідної мови. Якщо вихідна програма неправильна (містить хоча б одну помилку), то результатом роботи транслятора буде повідомлення про помилку. У цьому сенсі транслятор схожий на перекладача, наприклад, з англійської, якому підсунули невірний текст.

9.1.2 Визначення компілятора. Відмінність компілятора від транслятора

Компілятор - це транслятор, який здійснює переклад вихідної програми в еквівалентну їй результуючу програму мовою машинних команд або мовою асемблера.

Таким чином, компілятор відрізняється від транслятора лише тим, що його результуюча програма завжди має бути написана ***мовою машинних кодів*** або ***мовою асемблера***. Результуюча програма транслятора, у загальному випадку, може бути написана будь-якою мовою - можливий, наприклад, транслятор програм з мови Pascal на мову C.

Усякий компілятор є транслятором, але не навпаки - не всякий транслятор буде компілятором. Наприклад, згаданий вище транслятор з мови Pascal на C компілятором не є.

Результуюча програма компілятора називається ***об'єктною програмою***, або ***об'єктним кодом***, а вихідну програму в цьому випадку часто називають ***вихідним кодом***. Файл, у який записано об'єктну програму, зазвичай називають ***об'єктним файлом***. Навіть у тому випадку, коли результуюча програма породжується мовою машинних команд, між об'єктною програмою (об'єктним файлом) та виконуваною програмою (виконуваним файлом) є суттєва різниця. Породжена компілятором програма не може безпосередньо виконуватися на комп'ютері.

Саме слово "компілятор" походить від англійського терміна "compiler" ("укладач", "компонувальник"). Термін завдячує своєму походженню здатності компіляторів складати об'єктну програму з фрагментів машинних кодів, що відповідають синтаксичним конструкціям вихідної програми. Оскільки спочатку компілятори нічого іншого робити не вміли, то цей термін і закріпився за ними.

Результуюча програма, створена компілятором, будується мовою машинних кодів або асемблера, тобто мовами, які обов'язково орієнтовані на певну обчислювальну систему. Отже, така результуюча програма завжди призначена для виконання на обчислювальній системі з певною архітектурою.

Обчислювальна система, на якій виконується результуюча (об'єктна)

програма, створена компілятором, називається ***цільовою обчислювальною системою.***

У поняття цільової обчислювальної системи входять такі компоненти:

архітектура апаратних засобів комп'ютера,

операційна система,

набір бібліотек, що динамічно підключаються, які необхідні для виконання об'єктної програми.

При цьому слід пам'ятати, що об'єктна програма орієнтована на цільову обчислювальну систему, але не може бути безпосередньо виконана на ній без додаткової обробки.

Цільова обчислювальна система не завжди є тією самою обчислювальною системою, на якій працює сам компілятор. Часто вони збігаються, але буває так, що компілятор працює під керуванням обчислювальної системи одного типу, а будує об'єктні програми, призначені для виконання на обчислювальних системах зовсім іншого типу.

Компілятори, безумовно, найпоширеніший вид трансляторів. Вони мають найширше практичне застосування, яким завдячують широкому поширенню всіляких мов програмування.

Природно, транслятори і компілятори, як і всі інші програми, розробляють люди - зазвичай це група розробників. У принципі, вони могли б створювати його безпосередньо на мові машинних команд, однак обсяг коду і даних сучасних компіляторів такий, що їх створення на мові машинних команд практично неможливе в розумні терміни за розумних трудовитрат. Тому практично всі сучасні компілятори також створюються за допомогою компіляторів (найчастіше в цій ролі виступають попередні версії компіляторів тієї ж фірми-виробника). І в цій якості компілятор є результуючою програмою для іншого компілятора, яка нічим не відрізняється від усіх інших породжуваних результуючих програм.

Наведемо приклади компіляторів. Компілятори з таких мов, як FORTRAN, ALGOL-68, PL/1, були орієнтовані на великі ЕОМ з пакетною обробкою завдань. Мови ADA, Modula, Simula відомі лише вузькому колу фахівців і не набули широкого поширення. Водночас на ринку програмних систем домінують системи програмування і компілятори для мов, яким не пророкували світлого майбутнього. Насамперед, зараз це C і C++. Ще можна згадати доволі поширений Pascal, який несподівано для багатьох вийшов за рамки суто навчальної мови для університетського середовища.

9.1.3 Визначення інтерпретатора. Різниця інтерпретаторами та трансляторами

Крім схожих між собою понять "транслятор" і "компілятор" існує принципово відмінне від них поняття інтерпретатора.

Інтерпретатор - це програма, яка сприймає вихідну програму вхідною (вихідною) мовою та виконує її.

Інтерпретатор, так само як і транслятор, аналізує текст вихідної програми. Різниця полягає в тому, що він ***не породжує результуючу програму***, а одразу ж виконує вихідну відповідно до її змісту, заданого семантикою вхідної мови. Таким чином, результатом роботи інтерпретатора буде результат, визначений смислом вихідної програми, у тому разі, якщо ця програма синтаксично та семантично правильна з точки зору вхідної мови програмування, або повідомлення про помилку в іншому разі.

Щоб виконати вихідну програму, інтерпретатор так чи інакше має перетворити її на мову машинних кодів, оскільки інакше виконання програм на комп'ютері неможливе. Він, звісно ж, робить це, однак отримані машинні коди не є доступними - їх не бачить користувач інтерпретатора. Ці машинні коди породжуються інтерпретатором, виконуються і знищуються в міру потреби - так, як того вимагає конкретна реалізація інтерпретатора. Користувач же бачить результат виконання цих кодів - тобто результат виконання вихідної програми (вимога про еквівалентність вихідної програми і породжених машинних кодів і в цьому разі, безумовно, має виконуватися).

Далеко не всі мови програмування допускають побудову інтерпретаторів. Для цього мова повинна допускати можливість існування компілятора, що виконує розбір вихідної програми за один прохід. Крім того, мова не може інтерпретуватися в міру надходження команд, якщо вона допускає появу звернень до функцій і структур даних раніше їх безпосереднього опису. Тому таким методом не можуть інтерпретуватися такі мови, як C і Pascal.

Відсутність кроку оптимізації призводить до того, що виконання програми за допомогою інтерпретатора є менш ефективним, ніж за допомогою аналогічного компілятора.

Перевагою інтерпретатора є незалежність виконання програми від архітектури цільової обчислювальної системи. Для інтерпретації програми необхідно мати лише її вихідний текст та інтерпретатор з відповідної мови.

З розвитком глобальних мереж і поширенням всесвітньої мережі Інтернет з'явилося багато нових систем, що інтерпретують текст вихідної програми.

З відомих мов, які передбачали інтерпретацію, можна згадати Basic, хоча більшості зараз відома його компілювальна реалізація Visual Basic, зроблена фірмою Microsoft. Проте зараз ситуація дещо змінилася, оскільки питання про переносимість програм та їхню апаратно-платформну незалежність набуває дедалі більшої актуальності з розвитком мережі Інтернет.

Прикладом інтерпретованої мови може слугувати HTML (Hypertext

Markup Language) - мова опису гіпертексту. На її основі нині функціонує практично вся структура мережі Інтернет.

Інший приклад - мови Java і JavaScript поєднують у собі функції компіляції та інтерпретації (текст вихідної програми компілюють у деякий проміжний код, не залежний від архітектури цільової обчислювальної системи, цей код поширюють мережею, а потім його інтерпретують на стороні, що приймає).

9.1.4 Етапи трансляції. Загальна схема роботи транслятора

На рис. 27 наведено загальну схему роботи компілятора, з якої видно, що загалом процес компіляції складається з двох основних етапів - аналізу та синтезу.

На етапі аналізу виконується таке:

розпізнавання тексту вихідної програми

створення та заповнення таблиць ідентифікаторів.

Результатом його роботи слугує якесь внутрішнє уявлення програми, зрозуміле компілятору.

На етапі синтезу на підставі внутрішнього представлення програми та інформації, що міститься в таблиці ідентифікаторів, породжується текст результуючої програми. Результатом цього етапу є об'єктний код.

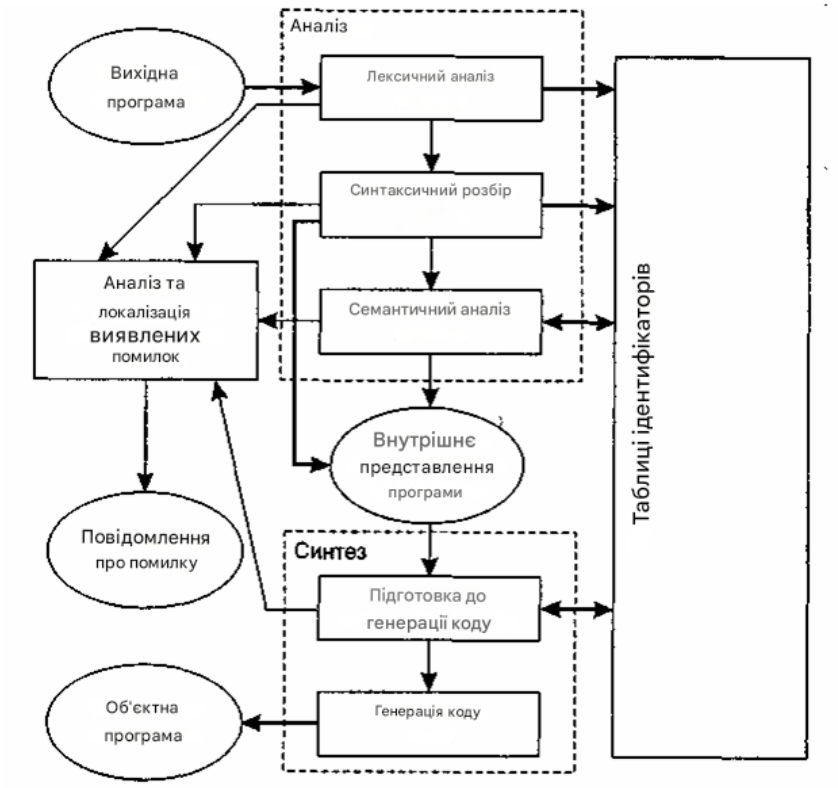

Рис. 27 − Загальна схема роботи компілятора

Крім того, у складі компілятора ***є частина, відповідальна за аналіз і виправлення помилок***, яка в разі наявності помилки в тексті вихідної програми повинна максимально повно інформувати користувача про тип помилки та місце її виникнення. У кращому разі компілятор може запропонувати користувачеві варіант виправлення помилки.

Ці етапи, своєю чергою, складаються з дрібніших етапів, які називаються ***фазами компіляції***. Склад фаз компіляції на мал. 27 наведено в найзагальнішому вигляді, їхня конкретна реалізація і процес взаємодії можуть, звісно, відрізнятися залежно від версії компілятора. Однак у тому чи іншому вигляді всі представлені фази практично завжди присутні в кожному конкретному компіляторі.

Компілятор загалом з погляду теорії формальних мов виконує такі ***основні функції***:

1. Компілятор є ***розпізнавачем*** для мови вихідної програми. Тобто він має отримати на вхід ланцюжок символів вхідної мови, перевірити його приналежність до мови і, більше того, виявити правила, за якими цей ланцюжок було побудовано (оскільки на запитання про приналежність сама відповідь "так" чи "ні" становить мало інтересу). Цікаво, що генератором ланцюжків вхідної мови виступає користувач - автор вихідної програми.

2. Компілятор є ***генератором*** для мови результуючої програми. Він має побудувати на виході ланцюжок вихідної мови за певними правилами, передбачуваними мовою машинних команд або мовою асемблера. У разі машинних команд розпізнавачем цього ланцюжка виступатиме цільова обчислювальна система, під яку створюється результуюча програма.

Розглянемо перелік основних фаз (частин) компіляції та короткий опис їхніх функцій.

Лексичний аналіз (сканер) - це частина компілятора, яка читає літери програми вихідною мовою і будує з них слова (лексеми) вихідної мови. Робота його полягає в такому:

на вхід лексичного аналізатора надходить текст вихідної програми;

вихідна інформація передається для подальшої обробки компілятором на етапі синтаксичного розбору.

З теоретичної точки зору лексичний аналізатор не є обов'язковою, необхідною частиною компілятора. Однак існують причини, які визначають його присутність практично у всіх компіляторах.

Синтаксичний розбір - це основна частина компілятора на етапі аналізу. Вона призначена для наступного:

виконується виділення синтаксичних конструкцій у тексті вихідної програми, обробленому лексичним аналізатором;

перевіряється синтаксична правильність програми.

Синтаксичний розбір відіграє головну роль - роль розпізнавача тексту вхідної мови програмування.

Семантичний аналіз - це частина компілятора, що перевіряє правильність тексту вихідної програми з погляду семантики вхідної мови, а також виконує перетворення тексту, які вимагає семантика вхідної мови (наприклад, такі, як додавання функцій неявного перетворення типів).

У різних реалізаціях компіляторів семантичний аналіз може частково входити у фазу синтаксичного розбору, частково - у фазу підготовки до генерації коду.

Підготовка до генерації коду - це фаза, на якій компілятор виконує попередні дії, безпосередньо пов'язані із синтезом тексту результуючої програми, але які ще не призводять до породження тексту вихідною мовою. Зазвичай до цієї фази входять дії, пов'язані з ідентифікацією елементів мови, розподілом пам'яті тощо.

Генерація коду - це фаза, безпосередньо пов'язана з породженням команд, що складають речення вихідної мови і загалом текст результуючої програми. Це основна фаза на етапі синтезу результуючої програми.

Крім безпосереднього породження тексту результуючої програми генерація зазвичай містить у собі також ***оптимізацію*** - процес, пов'язаний з обробкою вже породженого тексту. Іноді оптимізацію виокремлюють в окрему фазу компіляції, оскільки вона істотно впливає на якість і ефективність результуючої програми.

Таблиці ідентифікаторів - це спеціальним чином організовані набори даних, які слугують для зберігання інформації про елементи вихідної програми, які потім використовуються для породження тексту результуючої програми.

У конкретній реалізації компілятора може бути як одна, так і кілька таблиць ідентифікаторів. Елементами вихідної програми, інформацію про які необхідно зберігати в процесі компіляції, є змінні, константи, функції тощо. - конкретний склад набору елементів залежить від використовуваної вхідної мови програмування.

Поняття "таблиці" зовсім не передбачає, що це сховище даних має бути організовано саме у вигляді таблиць або інших масивів інформації.

Представлений на рис. 26 розподіл процесу компіляції на фази слугує скоріше методичним цілям і на практиці може не дотримуватися настільки суворо.

Далі розглянемо ***загальні аспекти взаємозв'язку фаз компіляції***:

1. На фазі лексичного аналізу лексеми виділяються з тексту вхідної програми остільки, оскільки вони необхідні для наступної фази синтаксичного розбору.

2. Синтаксичний розбір і генерація коду можуть виконуватися одночасно.

Таким чином, ці три фази компіляції можуть працювати комбіновано, а разом з ними може виконуватися і підготовка до генерації коду.

9.1.5 Поняття проходу. Багатопрохідні та однопрохідні компілятори

Для розгляду технічних питань реалізації основних фаз компіляції, необхідно познайомитися з поняттям проходу.

Реальні компілятори, як правило, виконують трансляцію тексту вихідної програми за кілька проходів.

Прохід - це процес послідовного читання компілятором даних із зовнішньої пам'яті, їхнього опрацювання та поміщення результату роботи в зовнішню пам'ять.

Найчастіше один прохід включає в себе виконання однієї або декількох фаз компіляції. Результатом проміжних проходів є внутрішнє представлення вихідної програми, результатом останнього проходу - об'єктна програма.

Як зовнішня пам'ять можуть виступати будь-які носії інформації:

- оперативна пам'ять комп'ютера;
- накопичувачі на магнітних дисках;
- накопичувачі на магнітних стрічках тощо.

Сучасні компілятори, як правило, прагнуть максимально використовувати для зберігання даних оперативну пам'ять (ОП) комп'ютера, і тільки в разі нестачі обсягу доступної пам'яті використовуються накопичувачі на жорстких магнітних дисках. Інші носії інформації в сучасних компіляторах не використовуються через невисоку швидкість обміну даними.

Під час виконання кожного проходу компілятору доступна інформація, отримана в результаті всіх попередніх проходів. Як правило, він прагне використовувати насамперед тільки інформацію, отриману на проході, що безпосередньо передував поточному, але, в принципі, може звертатися і до даних від більш ранніх проходів аж до вихідного тексту програми.

Інформація, одержувана компілятором під час виконання проходів, недоступна користувачеві. Вона або зберігається в ОП, яка звільняється компілятором після завершення процесу трансляції, або оформляється у вигляді тимчасових файлів на диску, які також знищуються після завершення роботи компілятора.

Тому людина, яка працює з компілятором, може навіть не знати, скільки проходів виконує компілятор, - вона завжди бачить тільки текст вихідної програми і результуючу об'єктну програму. Але кількість виконуваних проходів - це важлива технічна характеристика компілятора, солідні фірми-розробники компіляторів зазвичай вказують її в описі свого продукту.

Зрозуміло, що розробники прагнуть максимально скоротити кількість

проходів, виконуваних компіляторами. При цьому збільшується швидкість роботи компілятора, скорочується обсяг необхідної йому пам'яті.

Однопрохідний компілятор, який отримує на вхід вихідну програму і відразу ж породжує результуючу об'єктну програму, - це ідеальний варіант.

Однак скоротити кількість проходів не завжди вдається. Кількість необхідних проходів визначається насамперед граматикою і семантичними правилами вихідної мови. Що складніша граматика мови і що більше варіантів припускають семантичні правила, то більше проходів виконуватиме компілятор (звісно, відіграє свою роль і кваліфікація розробників компілятора). Наприклад, саме тому зазвичай компілятори з мови Pascal працюють швидше, ніж компілятори з мови С - граматика Pascal простіша, а семантичні правила жорсткіші.

Реальні компілятори виконують, як правило, від двох до п'яти проходів. Таким чином, реальні компілятори є ***багатопрохідними***.

Найпоширенішими є дво- і трипрохідні компілятори, наприклад ті, що виконують такі проходи:

- лексичний аналіз;
- синтаксичний розбір і семантичний аналіз;
- генерацію та оптимізацію коду.

У сучасних системах програмування нерідко перший прохід компілятора (лексичний аналіз коду) виконується паралельно з редагуванням коду вихідної програми.

9.2 Таблиці ідентифікаторів. Організація таблиць ідентифікаторів

9.2.1 Призначення таблиць ідентифікаторів

Під час виконання семантичного аналізу, генерації коду та оптимізації результуючої програми компілятор має оперувати характеристиками основних елементів вихідної програми - змінних, констант, функцій та інших лексичних одиниць вхідної мови. Ці характеристики можуть бути отримані компілятором на етапі синтаксичного аналізу вхідної програми (найчастіше під час аналізу структури блоків описів змінних і констант), а також доповнені на етапі підготовки до генерації коду (наприклад, під час розподілу пам'яті).

Набір характеристик, що відповідає кожному елементу вихідної програми, залежить від типу цього елемента, від його сенсу (семантики) і, відповідно, від тієї ролі, яку він виконує у вихідній і результуючій програмах. У кожному конкретному випадку цей набір характеристик може бути свій залежно від синтаксису і семантики вхідної мови, від архітектури цільової обчислювальної системи і від структури компілятора. Але є типові характеристики, які найчастіше притаманні тим чи іншим елементам вихідної програми.

Змінні, константи, функції та інші елементи у вихідній програмі, як правило, позначаються ***ідентифікаторами***.

Головною характеристикою будь-якого елемента вихідної програми є його ***ім'я***. Саме з іменами змінних, констант, функцій та інших елементів вхідної мови оперує розробник програми - тому і компілятор має вміти аналізувати ці елементи за їхніми іменами.

Ім'я кожного елемента має бути ***унікальним***. Багато сучасних мов програмування допускають збіги (неунікальність) імен змінних і функцій залежно від їхньої області видимості та інших умов вихідної програми. У цьому випадку унікальність імен повинен забезпечувати сам компілятор.

Розглянемо, що відбувається з ідентифікаторами на різних фазах трансляції (табл. 16).

Таблиця 16 − Дії з ідентифікатором на різних фазах трансляції

Фаза компіляції	Дія з ідентифікатором
1. Лексичний аналіз	виділення
2. Синтаксичний розбір	визначення характеристик
3. Семантичний аналіз	
4. Підготовка до генерації коду	

Компілятор повинен мати можливість зберігати всі знайдені ідентифікатори та пов'язані з ними характеристики протягом усього процесу компіляції.

У компіляторах використовуються спеціальні сховища даних, які називаються ***таблицями ідентифікаторів***.

Компілятор може працювати з однією або кількома таблицями ідентифікаторів - їхня кількість залежить від реалізації компілятора. Наприклад, можна організовувати різні таблиці ідентифікаторів для різних модулів вихідної програми або для різних типів елементів вхідної мови.

Склад інформації, що зберігається в таблиці ідентифікаторів для кожного елемента вихідної програми, залежить від семантики вхідної мови і типу елемента.

Наприклад, у таблицях ідентифікаторів може зберігатися така інформація:

- для змінних:
- ім'я змінної;
- тип даних змінної;
- область пам'яті, пов'язана зі змінною;
- для констант:
- назва константи (якщо вона є);
- значення константи;
- тип даних константи (якщо потрібно);

- для функцій:
- ім'я функції;
- кількість і типи формальних аргументів функції;
- тип результату, що повертається;
- адреса коду функції.

Наведений вище склад збереженої інформації є тільки приблизним. Конкретне наповнення таблиць ідентифікаторів залежить від реалізації компілятора.

Принцип роботи компілятора з таблицею ідентифікаторів такий: на різних фазах компіляції він змушений багаторазово звертатися до таблиці для пошуку інформації та запису нових даних.

Компілятору доводиться виконувати пошук необхідного елемента в таблиці ідентифікаторів на ім'я частіше, ніж поміщати новий елемент у таблицю, тому що кожен ідентифікатор може бути описаний тільки один раз, а використаний - кілька разів.

Звідси можна зробити висновок, що таблиці ідентифікаторів мають бути організовані таким чином, щоб компілятор мав можливість максимально швидкого пошуку потрібного йому елемента.

9.2.2 Принципи організації таблиць ідентифікаторів

Компілятор поповнює записи в таблиці ідентифікаторів у міру аналізу вихідної програми і виявлення в ній нових елементів, які потребують розміщення в таблиці. Пошук інформації в таблиці виконується щоразу, коли компілятору необхідні відомості про той чи інший елемент програми. Причому слід зауважити, що пошук елемента в таблиці буде виконуватися компілятором істотно частіше, ніж розміщення в неї нових елементів. Так відбувається тому, що описи нових елементів у вихідній програмі, як правило, зустрічаються набагато рідше, ніж ці елементи використовуються. Крім того, кожному додаванню елемента в таблицю ідентифікаторів у будь-якому разі передуватиме операція пошуку - щоб переконатися, що такого елемента в таблиці немає.

На кожну операцію пошуку елемента в таблиці компілятор витрачатиме час, і оскільки кількість елементів у вихідній програмі велика (від одиниць до сотень тисяч залежно від об'єму програми), цей час суттєво впливатиме на загальний час компіляції. Тому таблиці ідентифікаторів мають бути організовані таким чином, щоб компілятор мав можливість максимально швидко виконувати пошук потрібного йому запису таблиці за ім'ям елемента, з яким пов'язаний цей запис.

Можна виділити такі способи організації таблиць ідентифікаторів:

- прості та впорядковані списки;
- бінарне дерево;

- хеш-адресація з рехешуванням;
- хеш-адресація за методом ланцюжків;
- комбінація хеш-адресації зі списком або бінарним деревом.

9.2.3 Найпростіші методи побудови ідентифікаторів

У найпростішому випадку таблиця ідентифікаторів являє собою лінійний невпорядкований список, або масив, кожна комірка якого містить дані про відповідний елемент таблиці. Розміщення нових елементів у такій таблиці виконується шляхом запису інформації в чергову комірку масиву або списку в міру виявлення нових елементів у вихідній програмі.

Пошук потрібного елемента в таблиці буде в цьому випадку виконуватися шляхом послідовного перебору всіх елементів і порівняння їхнього імені з ім'ям шуканого елемента, доки не буде знайдений елемент з таким самим ім'ям. Тоді якщо за одиницю часу прийняти час, що витрачається компілятором на порівняння двох рядків (у сучасних обчислювальних системах таке порівняння найчастіше виконується однією командою), то для таблиці, що містить **N** елементів, у середньому буде виконано **N/2** порівнянь.

Час, необхідний на додавання нового елемента в таблицю ($\mathbf{T_д}$), не залежить від числа елементів у таблиці (**N**). Але якщо **N** велике, то пошук потребуватиме значних витрат часу. Час пошуку ($\mathbf{T_п}$) у такій таблиці можна оцінити як $\mathbf{T_п = O(N).}$ Оскільки саме пошук у таблиці ідентифікаторів є найбільш часто виконуваною компілятором операцією, такий спосіб організації таблиць ідентифікаторів є неефективним. Він може бути застосований тільки для найпростіших компіляторів, які працюють з невеликими програмами.

Пошук може бути виконаний більш ефективно, якщо елементи таблиці відсортовані (впорядковані) природним чином. Оскільки пошук здійснюється за іменем, найприроднішим рішенням буде розташувати елементи таблиці в прямому або зворотному алфавітному порядку. Ефективним методом пошуку в упорядкованому списку з N елементів є ***бінарний***, або ***логарифмічний, пошук***.

Алгоритм логарифмічного пошуку полягає в такому:

- шуканий символ порівнюється з елементом **(N+ 1)/2** у середині таблиці;
- якщо цей елемент не є шуканим, то ми маємо переглянути тільки блок елементів, пронумерованих від 1 до **(N+1)/2 - 1**, або блок елементів від **(N+ 1)/2 + 1** до **N** залежно від того, менший або більший від шуканого елемента, ніж той, з яким його порівняли;
- потім процес повторюється над потрібним блоком удвічі меншого розміру;
- так триває доти, доки або шуканий елемент не буде знайдений, або

алгоритм не дійде до чергового блоку, що містить один або два елементи (з якими можна виконати пряме порівняння шуканого елемента).

Оскільки на кожному кроці число елементів, які можуть містити шуканий елемент, скорочується вдвічі, максимальне число порівнянь дорівнює $\mathbf{1 + \log_2 N}$. Тоді час пошуку елемента в таблиці ідентифікаторів можна оцінити як $\mathbf{T_п = O (\log_2 N)}$

Для порівняння: за N = 128 бінарний пошук потребує щонайбільше 8 порівнянь, а пошук у невпорядкованій таблиці - у середньому 64 порівняння.

Метод називають ***"бінарним пошуком"***, оскільки на кожному кроці обсяг інформації, що розглядається, скорочується вдвічі, а ***"логарифмічним"*** - оскільки час, що витрачається на пошук потрібного елемента в масиві, має логарифмічну залежність від загальної кількості елементів у ньому.

Недоліком логарифмічного пошуку є вимога впорядкування таблиці ідентифікаторів. Оскільки масив інформації, у якому виконується пошук, має бути впорядкований, час його заповнення вже залежатиме від кількості елементів у масиві. Таблиця ідентифікаторів часто проглядається компілятором ще до того, як вона заповнена, тому потрібно, щоб умова впорядкованості виконувалася на всіх етапах звернення до неї. Отже, для побудови такої таблиці можна користуватися тільки алгоритмом прямого впорядкованого включення елементів.

Якщо користуватися стандартними алгоритмами, застосовуваними для організації впорядкованих масивів даних, то середній час, необхідний на поміщення всіх елементів у таблицю, можна оцінити так:

$$\mathbf{T_д = O (N * \log_2 N) + k * O (N^2)} \qquad (1)$$

Тут **k** - деякий коефіцієнт, що відображає співвідношення між часами, витраченими комп'ютером на виконання операції порівняння та операції перенесення даних.

При організації логарифмічного пошуку в таблиці ідентифікаторів забезпечується істотне скорочення часу пошуку потрібного елемента за рахунок збільшення часу на поміщення нового елемента в таблицю. Оскільки додавання нових елементів у таблицю ідентифікаторів відбувається істотно рідше, ніж звернення до них, цей метод слід визнати більш ефективним, ніж метод організації невпорядкованої таблиці. Однак у реальних компіляторах цей метод безпосередньо також використовується рідко, оскільки існують більш ефективні методи.

9.2.4 Побудова таблиць ідентифікаторів за методом бінарного дерева

Можна скоротити час пошуку шуканого елемента в таблиці ідентифікаторів, не збільшуючи значно час, необхідний на її заповнення. Для цього треба відмовитися від організації таблиці у вигляді безперервного масиву даних. Існує метод побудови таблиць, за якого таблиця має форму бінарного дерева. Кожен вузол дерева являє собою елемент таблиці, причому кореневим вузлом стає перший елемент, зустрінутий компілятором під час заповнення таблиці. Дерево називається бінарним, оскільки кожна вершина в ньому може мати не більше двох гілок. Для визначеності будемо називати дві гілки "права" і "ліва".

Розглянемо ***алгоритм заповнення бінарного дерева***. Будемо вважати, що алгоритм працює з потоком вхідних даних, який містить ідентифікатори. Перший ідентифікатор поміщається у вершину дерева. Усі подальші ідентифікатори потрапляють у дерево за таким алгоритмом

.Вибрати черговий ідентифікатор із вхідного потоку даних. Якщо чергового ідентифікатора немає, то побудову дерева закінчено.

Зробити поточним вузлом дерева кореневу вершину.

Порівняти ім'я чергового ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.

Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, якщо дорівнює - припинити виконання алгоритму (двох однакових ідентифікаторів бути не повинно), інакше - перейти до кроку 7.

Якщо в поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше - перейти до кроку 6.

Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину лівою вершиною поточного вузла і повернутися до кроку 1.

Якщо в поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше - перейти до кроку 8.

Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину правою вершиною поточного вузла і повернутися до кроку 1.

Розглянемо як приклад послідовність ідентифікаторів GA, Dl, M22, E, A12, BC, F. На рис. 28 проілюстровано весь процес побудови бінарного дерева для цієї послідовності ідентифікаторів.

Пошук елемента в дереві виконується за алгоритмом, схожим з алгоритмом заповнення дерева:

Зробити поточним вузлом дерева кореневу вершину.

Порівняти ім'я шуканого ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.

Якщо імена збігаються, то шуканий ідентифікатор знайдено, алгоритм

завершується, інакше треба перейти до кроку 4.

Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, інакше - перейти до кроку 6.

Якщо у поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий ідентифікатор не знайдено, алгоритм завершується.

Якщо у поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий ідентифікатор не знайдено, алгоритм завершується.

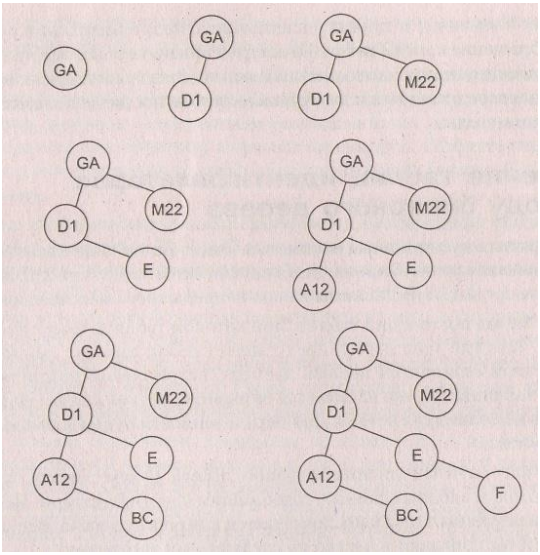

Рис. 28 − Приклад заповнення бінарного дерева

Для цього методу кількість необхідних порівнянь і форма отриманого дерева залежать від того порядку, в якому надходять ідентифікатори. Наприклад, якщо в розглянутому вище прикладі замість послідовності ідентифікаторів GA, D1, M22, E, A12, BC, F узяти послідовність A12, BC, D1, E, F, GA, M22, то дерево виродиться в упорядкований односпрямований зв'язний список. Ця особливість є недоліком цього методу організації таблиць ідентифікаторів. Іншими недоліками методу є: необхідність зберігати два додаткові посилання на ліву і праву гілки в кожному елементі дерева і робота з динамічним виділенням пам'яті під час побудови дерева.

Якщо припустити, що послідовність ідентифікаторів у вихідній програмі є статистично невпорядкованою (що загалом відповідає дійсності), то можна вважати, що побудоване бінарне дерево буде невиродженим. Тоді середній час на заповнення дерева ($T_д$) і на пошук елемента в ньому ($T_п$) можна оцінити таким чином:

$$T_д = N * O(\log_2 N) \quad (2)$$

$$T_п = O(\log_2 N) \quad (3)$$

Незважаючи на зазначені недоліки, метод бінарного дерева є досить

вдалим механізмом для організації таблиць ідентифікаторів. Він знайшов своє застосування в низці компіляторів. Іноді компілятори будують кілька різних дерев для ідентифікаторів різних типів і різної довжини.

Контрольні питання

1. Що таке транслятор і які основні його компоненти?
2. Які три програми беруть участь у роботі транслятора?
3. Що означає еквівалентність вихідної та результуючої програм у роботі транслятора?
4. Чому процес трансляції можна порівняти з перекладом між природними мовами?
5. Який результат роботи транслятора у разі наявності синтаксичних або семантичних помилок у вихідній програмі?
6. Які основні відмінності між компілятором і транслятором?
7. Чому важливо правильно вибрати вхідну та вихідну мови при створенні транслятора?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Русанова О. В., Корочкін О. В. Програмне забезпечення комп’ютерних систем. Програмування та компіляція. Київ : КПІ ім. Ігоря Сікорського, 2020. 94 с.

2. Тарарака В. Д. Архітектура комп'ютерних систем : навч. посібник. Житомир : ЖДТУ, 2018. 383 с.

3. Авраменко В. С., Авраменко А. С. Проектування інформаційних систем : навч. посібник. Черкаси : Черкас. нац. ун-т ім. Б. Хмельницького, 2017. 434 с.

4. Погребняк Б. І., Булаєнко М. В. Операційні системи : навч. посібник / Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. Харків : ХНУМГ ім. О. М. Бекетова, 2018. 104 с.

5. Зайцев В. Г., Дробязко І. П. Операційні системи : навч. посібник [Електронний ресурс] / КПІ ім. Ігоря Сікорського. Електронні текстові дані (1 файл : 3 Мбайт). Київ : КПІ ім. Ігоря Сікорського, 2019. 240 с.

6. Архітектура комп’ютерних систем : конспект лекцій для студентів усіх форм навчання з курсу «Архітектура комп’ютерних систем» / уклад. О. С. Голотенко. Тернопіль : Вид-во ТНТУ ім. І. Пулюя, 2016. 120 с.

7. Батрак Є. О. Архітектура комп'ютерних систем : лабораторний практикум. Київ : КПІ ім. Ігоря Сікорського, 2020. 110 с.

8. Руденко В. М., Чегренець І. В. Комп’ютерна логіка та архітектура комп’ютерних систем : навч. посібник. Київ : Держ. ун-т інформац.-комунікац. технологій, 2019. 180 с.

9. Антоненко О. В., Бардус І. О. Архітектура комп'ютера та конфігурування комп'ютерних систем : навч. посібник. Дніпро : Дніпропетр. держ. техн. ун-т, 2018. 200 с.

Навчальне видання

КОМП'ЮТЕРНІ СИСТЕМИ

Курс лекцій

ЧУБ Ірина

Формат 60×84/16. Гарнітура Times New Roman Папір для цифрового друку. Друк ризографічний.
Ум. друк. арк. 2,67. Наклад 20 пр.
ДБТУ
61002, м. Харків, вул. Алчевських, 44