МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

В.Г. Зайцев, І.П. Дробязко

ОПЕРАЦІЙНІ СИСТЕМИ

Рекомендовано методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальніст
ю 123 "Комп'ютерна інженерія"

Київ

«КПІ ім. Ігоря Сікорського»

2019

Рецензенти: *Забара С. С.*, д-р техн. наук, проф.
Савченко Ю.Г., д-р техн. наук, проф.

Відповідальний редактор *Тарасенко В. П.*, д-р техн. наук, проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол №11 від 20.06.2019 р.)
за поданням Вченої ради факультету прикладної математики (протокол № 10 від 30.05.2019 р.)

Електронне мережне навчальне видання

Зайцев Володимир Григорович, д-р техн. наук, проф.
Дробязко Ірина Павлівна, ст. викл.

ОПЕРАЦІЙНІ СИСТЕМИ
КОНСПЕКТ ЛЕКЦІЙ

Операційні системи: [Електронний ресурс]: навч. посіб. для студ. спеціальності 123 «Комп'ютерна інженерія» / В. Г. Зайцев, І. П. Дробязко; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 3 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2019. – 240 с.

Навчальний посібник призначений для отримання знань студентами, що вивчають дисципліну «Операційні системи», щодо типових підходів до побудови і особливостей функціонування основних компонентів операційних систем (ОС).

Навчальне видання призначене для студентів, які навчаються за спеціальністю 123 «Комп'ютерна інженерія» факультету прикладної математики КПІ ім. Ігоря Сікорського.

ЗМІСТ

ВСТУП

Операційна система (ОС) комп'ютера – комплекс взаємозв'язаних програм, що виконують функції інтерфейсу між додатками і користувачем, з одного боку, та апаратурою комп'ютера, з іншого. У відповідності до цього ОС виконує наступні основні функції:

- представлення користувачу або програмісту замість реальної апаратури комп'ютера розширеної віртуальної машини, з якою легко працювати та яку просто програмувати;
- підвищення ефективності використання комп'ютера шляхом раціонального управління його ресурсами у відповідності заданим критеріям.

Основною метою даного посібника є ознайомлення студентів з особливостями побудови і функціонування ОС та формування у студентів знань про:

- призначення та функції операційних систем;
- еволюцію операційних систем у зв'язку з розвитком обчислювальної техніки;
- архітектуру операційних систем;
- особливості мультипрограмування;
- методи планування процесів та потоків;
- засоби мультипрограмування на принципах переривань;
- основні засоби синхронізації процесів та потоків;
- проблему розподілу ресурсів та запобігання тупіків;
- алгоритми планування та планувальники;
- організацію та засоби управління пам'яттю;
- принципи побудови засобів введення-виведення інформації;
- файлові системи: логічну та фізичну організацію;
- порівняльний аналіз найбільш поширених методів організації сучасних файлових систем;

Обсяг і зміст посібника відповідають нормативним обсягам та змісту дисципліни, що вказані в освітньому стандарті спеціальності 123 «Комп'ютерна інженерія», і тому є мінімально достатніми для засвоєння їх студентами.

Для засвоєння матеріалу посібника необхідно мати математичну підготовку з дисципліни «Математичний аналіз» в обсязі, прийнятому для технічних університетів, а також програмістську підготовку з дисциплін «Програмування» в обсязі, передбаченому освітніми стандартами для спеціальності «Комп'ютерна інженерія».

Матеріали посібника підготовлені викладачами кафедри системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

Розділ 1. Основні функції операційних систем

Під операційною системою (ОС) звичайно розуміють комплекс керуючих та оброблюючих програм, що реалізують інтерфейс між апаратурою комп’ютера і користувачем, з одного боку, та оптимізують використання ресурсів обчислювальної системи при вирішенні конкретних задач, з іншого боку. ОС має виконувати наступні функції:

1. Забезпечувати завантаження користувальницьких програм в оперативну пам'ять й їхнє виконання.
2. Забезпечувати керування пам'яттю. У найпростішому випадку, це – вказівка єдиній завантажуваній програмі на кінцеву адресу пам'яті, доступної для використання, і на початкову адресу пам'яті, зайнятої системою.
3. Забезпечувати роботу із пристроями довгострокової пам'яті.
4. Надати стандартизований доступ до різних периферійних пристроїв.
5. Надати користувальницький інтерфейс.

У ряді ОС, функції цим вичерпуються (наприклад, у MS-DOS). Більш розвинені ОС надають додаткові можливості:

1. Паралельне (або псевдопаралельне) виконання декількох завдань.
2. Організація взаємодії завдань один з одним.
3. Організація максимальної взаємодії й поділу ресурсів.
4. Захист системних ресурсів, даних і програм користувача.
5. Аутентифікація користувача (перевірка користувача, чи він є тим, за кого себе видає, й має право виконати ту або іншу операцію).

1.1 Класифікація ОС

В залежності від того, які з перерахованих вище функцій операційні системи виконують, їх можна розділити на:

ДОС (Дискові операційні системи)

Це системи, які беруть на себе виконання тільки перших чотирьох функцій. Як правило, це резидентний набір програм, і не більш того. ДОС завантажує користувальницьку програму в пам’ять і передає їй керування, після чого програма робить із системою, що їй заманеться. При завершенні програми вважається “гарним тоном” залишити машину в такому стані, щоб ДОС змогла продовжити роботу. Якщо програма приводить машину до якогось іншого стану, ДОС нічим цьому перешкодити не може.

Характерний приклад – різні завантажувальні монітори. Такі системи працюють одночасно тільки з однією програмою.

MS DOS для IBM PC – прямий спадкоємець такого монітора.

Існування подібних систем обумовлено їхньою простотою й малих ресурсів для їх реалізації.

Ще одна причина їх можливого використання навіть на досить потужних машинах – вимога програмної сумісності з ранніми моделями того ж самого сімейства комп’ютерів.

ОС загального призначення

Це системи, що виконують всі перераховані функції, наприклад, IBM DOS й ОС/360 і наші аналоги ОС ЄС.

Ці ОС розраховані на інтерактивну роботу одного або декількох користувачів у режимі поділу часу, при не дуже жорстких вимогах своєчасної реакції системи на зовнішні події. У таких системах велика увага приділяється захисту самої системи, програмного забезпечення й користувальницьких даних від помилкових і зловмисних дій програм і користувачів.

До цього класу ставиться відома ОС Windows 2000, а також системи сімейства UNIX.

Системи віртуальних машин

Ці ОС допускають одночасну роботу декількох програм, але створюють при цьому для кожної програми ілюзію того, що машина перебуває в повному її розпорядженні, як і при роботі під керуванням ДОС.

Віртуальні машини – цінний засіб при розробці й тестуванні крос-платформних програм. Вони також використовуються для налагодження модулів ядра або самої операційної системи.

Для таких систем характерні високі накладні витрати й порівняно низька надійність. Тому вони рідко використовуються для промислового застосування. У системах віртуальних машин приділяється велика увага емуляції роботи апаратури.

Часто ці системи є підсистемами ОС загального призначення, наприклад підсистема W_0W в Windows NT, Windows 2000, емулятор RT-II в VAX тощо.

Системи реального часу

Ці системи призначені для полегшення розробки так званих програм **реального часу** – програм, які управляють некомп'ютерним устаткуванням, часто із жорсткими часовими обмеженнями.

Відмітною ознакою системи реального часу є здатність гарантувати певний час реакції. Важливо враховувати різницю між гарантованістю й просто високою продуктивністю й низькими накладними витратами. Далеко не всі алгоритми й технічні рішення, навіть й ті, які забезпечують відмінний середній час реакції, підходять для програм і ОС реального часу.

Типовими представниками цього класу систем є відомі системи OS-9 й OS-9000.

По суті, мультимедійні алгоритми (тобто потребуючі синхронізації зображення на екрані й звуку) також є системами реального часу. Розбіжність звуку й зображення фіксується людиною вже в межах 30 мсек.

На сьогодні існує кілька визначень систем реального часу (СРЧ) (real time operating systems (RTOS)), однак більшість з них суперечить одне одному.

Наведемо деякі з цих визначень для демонстрації різних поглядів на призначення і основні завдання СРЧ:

1. Системою реального часу називається система, в якій успішність роботи будь-якої програми залежить не тільки від її логічної правильності, а й від часу, за який вона отримала результат. Якщо часові обмеження не витримані, тоді фіксується збій в роботі систем.

Таким чином, часові обмеження повинні бути гарантовано витримані. Це вимагає від системи бути передбачуваною, тобто, незалежно від свого поточного стану і завантаженості, видавати потрібний результат за необхідний час. При цьому бажано, щоб система забезпечувала якомога більший відсоток використання наявних ресурсів. Прикладом завдання, де потрібна СРЧ, є управління роботом, що бере деталь зі стрічки конвеєра. Деталь рухається, і робот має лише невелике часове вікно, коли він може її взяти. Якщо він запізниться, то деталь вже не буде на потрібній ділянці конвеєра, і, отже, робота не буде зроблена, незважаючи на те, що робот знаходиться в правильному місці. Якщо він позиціонується раніше, то деталь ще не встигне під'їхати, і він заблокує їй шлях.

Іншим прикладом може бути космічний апарат, що знаходиться на автопілоті. Сенсорні серводатчики повинні постійно передавати в керуючий комп'ютер результати вимірювань. Якщо результат будь-якого вимірювання буде пропущено, то це може привести до неприпустимої невідповідності між реальним станом систем космічного апарату і інформацією про нього в керуючій програмі. Розрізняють жорсткі (hard) і слабкі (soft) вимоги реального часу. Якщо запізнення програми призводить до повного порушення роботи керованої системи, тоді говорять про жорсткі вимоги реального часу (жорсткі СРЧ).

Якщо ж запізнювання призводить тільки до втрати продуктивності, тоді говорять про слабкі вимоги реального часу (м'які СРЧ). Більшість програмного забезпечення орієнтоване на м'який реальний час, а завдання

хорошої СРЧ – забезпечити рівень безпечного функціонування системи, навіть якщо керуюча програма ніколи не закінчить своєї роботи.

2. Стандарт POSIX 1003.1 визначає СРЧ наступним чином: реальний час в операційних системах – це здатність операційної системи забезпечити необхідний рівень сервісу в заданий проміжок часу.

3. Іноді системами реального часу називають системи постійної готовності (on-line системи), або інтерактивні системи з достатнім часом реакції. Звичайно це роблять фірми-виробники з маркетингових міркувань. Якщо інтерактивну програму називають працюючою в реальному часі, то це означає, що вона встигає обробляти запити від людини, для якої затримка в сотні мілісекунд навіть непомітна.

4. Часто поняття «система реального часу» ототожнюють з поняттям «швидка система». Це не завжди правильно. Час затримки реакції СРЧ на подію вже не так і важливий (він може досягати декількох секунд). Головне, щоб цього часу було достатньо для конкретного додатку і гарантовано. Часто алгоритм з гарантованим часом роботи менш ефективний, ніж алгоритм, що такою властивістю не володіє. Наприклад, алгоритм «швидкого» сортування (quicksort) в середньому працює значно швидше багатьох інших алгоритмів сортування, але його гарантована оцінка складності значно гірша.

5. У багатьох важливих сферах виконання додатків СРЧ вводяться свої поняття «реального часу». Так, про процес цифрової обробки сигналу кажуть, що він йде в «реальному часі», якщо аналіз (при введенні) і / або генерація (при виведенні) даних може бути проведено за той самий час, що і аналіз та / або генерація тих самих даних без цифрової обробки сигналу. Наприклад, якщо при обробці аудіо даних потрібно 2,01 секунди для аналізу 2,00 секунди звуку, тоді це не процес реального часу. Якщо ж потрібно 1,99 секунди, тоді це – процес реального часу. Виходячи з вищезазначеного, визначення системи реального часу можна подати в наступній інтерпретації.

Визначення. Система реального часу реагує в передбачуваний час на непередбачувану появу зовнішніх подій.

Основні відмінності між ОС РЧ та звичайними ОС можна продемонструвати за допомогою табл. 1.1.

Таблиця 1.1. Відмінності ОС РЧ

	ОС РЧ	Звичайна ОС
Основна задача	Можливість реагувати на події, що надходять зовні	Оптимально розподілити ресурси комп'ютера між користувачами та задачами
Як позиціонується	Інструмент для створення конкретного програмно- апаратного комплексу	Сприймається користувачем як сукупність додатків, готових до використання
На що орієнтована	Обробка зовнішніх подій	Обробка дій користувача
Кому призначена	Кваліфікованому розробнику	Розробнику середньої кввліфікації

Засоби крос-розробки

Це алгоритми, що призначені для розробки програм у двохмашинній конфігурації. При цьому редагування, компіляція й налагодження виконуються на інструментальній машині, а потім скомпільований код завантажується в цільову машину.

Прикладами таких ОС є системи програмування мікроконтролерів Intel. Такі системи, як правило, містять:

- набір компіляторів й асемблерів, що працюють на інструментальній машині з "нормальною" ОС;
- бібліотеки, які виконують велику частину функцій ОС при роботі програми (крім функції завантаження);
- засоби налагодження.

Системи проміжних типів

Існують системи, які важко віднести до якогось одного з перерахованих типів. Найбільш відомими з таких систем є MS Windows 3.x й Windows 95. Вони, як ОС, використовують апаратні засоби процесора для захисту й віртуалізації пам'яті, і навіть забезпечують деяку подібність багатозадачності, але не захищають себе й програми від помилок інших програм (тобто поводяться, в цьому сенсі, як ДОС).

Сімейства операційних систем

Часто існує спадковість між різними ОС. Така спадковість обумовлена вимогами сумісності й переносимості прикладного ПЗ, крім того, запозиченню окремих вдалих рішень.

На основі такої спадковості можна побудувати “генеалогічні дерева” ОС.

Так, можна виділити кілька сімейств нині експлуатованих ОС, наприклад, сімейство UNIX, у тому числі ОС Linux, а також сімейств, які вже вимерли або вимирають, наприклад, системи для великих комп'ютерів IBM OS-360, IBM-Vm, OS/2.

1.2 Архітектура ОС

Найбільш загальним підходом до структуризації операційної системи є поділ усіх її модулів на дві групи:

1. Ядро – модулі, що виконують основні функції ОС.
2. Модулі, що виконують допоміжні функції.

Модулі ядра виконують:

- керування процесами,
- керування пам'яттю,
- керування пристроями введення-виведення.

Ядро виконує такі функції, як перемикання контекстів, завантаження / вивантаження сторінок пам'яті, обробку переривань. Ці функції недоступні для програм. Вони створюють для них **прикладне програмне середовище**.

Додатки звертаються до ядра із запитами – **системними викликами** – для виконання тих чи інших дій (наприклад, відкриття й читання файлу, виводу графіки, одержання системного часу й т.п.).

Функції ядра, які можуть викликатися додатками, утворюють **інтерфейс прикладного програмування.**

Функції, виконувані модулями ядра, є найчастіше використовуваними, тому швидкість їхнього виконання визначає продуктивність всієї системи в цілому. Саме тому більша частина модулів ядра є резидентною, тобто постійно перебуває в оперативній пам'яті.

Допоміжні модулі ОС оформлюються або у вигляді програм, або у вигляді бібліотек процедур. Оскільки частина модулів ОС виконуються як звичайні додатки (тобто в стандартному для даної ОС форматі), то часто буває складно провести чітку грань між ОС і додатками.

Рішення про те, чи є якась програма частиною ОС чи ні, приймає виробник ОС.

Допоміжні модулі ОС поділяють на наступні групи:

- утиліти – програми, що вирішують окремі завдання керування й супроводу комп'ютерної системи (наприклад, програми стискування дисків, архівування даних);
- системні оброблюючі програми – текстові й графічні редактори, компілятори, компонувальники, відладчики і т. п.;
- програми надання користувачеві додаткових послуг – спеціальний користувальницький інтерфейс, калькулятор і т.п.
- бібліотеки процедур – спрощують розробку програм (наприклад, бібліотека математичних функцій, функцій введення-виведення й т.п.).

Допоміжні модулі ОС звертаються до функцій ядра за допомогою **системних викликів**. Поділ операційної системи на ядро й модулі-додатки забезпечує легку розширюваність ОС (рис.1.1).

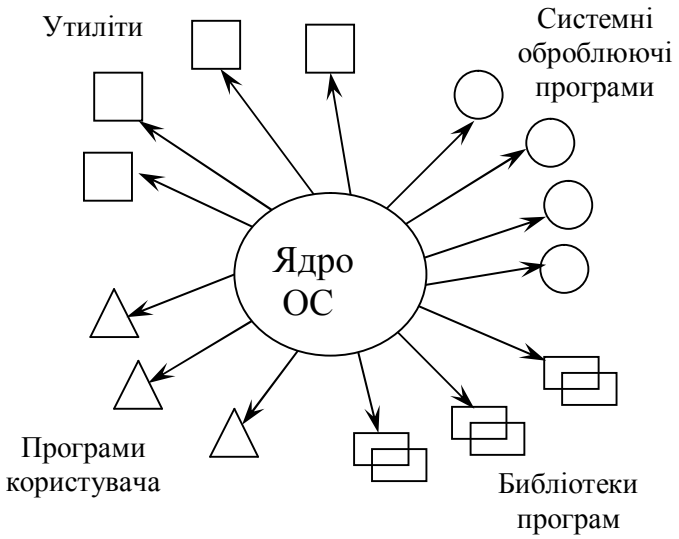


Рис. 1.1. Загальна структура ОС

Допоміжні модулі ОС завантажуються в оперативну пам'ять тільки на час свого виконання, тобто є **транзитними**.

Важливою властивістю архітектури, заснованої на ядрі, є можливість захисту кодів і даних операційної системи за рахунок виконання функцій ядра в привілейованому режимі.

Забезпечити привілеї ОС неможливо без спеціальних засобів апаратної підтримки. Апаратура комп'ютера повинна підтримувати як мінімум два режими роботи: користувальницький режим і привілейований режим, його також називають режимом ядра або режимом супервізора.

Додатки ставляться в підлеглий стан за рахунок заборони виконання в користувальницькому режимі деяких критичних команд, пов'язаних з перемиканням процесора із завдання на завдання, керуванням пристроями введення-виведення, доступом до механізмів розподілу й захисту пам'яті. Умова дозволу виконання критичних інструкцій перебуває під контролем ядра й забезпечується за рахунок набору інструкцій, безумовно, заборонених для користувальницького режиму.

Аналогічно забезпечуються привілеї ядра при доступі до пам'яті. Наприклад, виконання інструкції доступу до пам'яті для додатка

дозволяється, якщо інструкція звертається до області пам'яті, відведеної ОС даному додатку, і забороняється при звертанні до інших областей.

Між кількістю рівнів привілеїв, реалізованих апаратно, і кількістю рівнів привілеїв, підтримуваних ОС, немає прямої відповідності. Так, наприклад, на базі чотирьох рівнів, забезпечуваних процесорами Intel, ОС OS/2 будує трирівневую систему привілеїв, а Windows NT - дворівневу.

З іншого боку, якщо апаратура підтримує хоча б два рівні, програмним способом можна побудувати ОС із як завгодно розвиненою системою захисту. Розглянута архітектура ОС, заснована на привілейованому ядрі й додатках користувацького режиму, стала класичної. Її використовують багато відомих ОС: UNIX, VMS, OS/390, OS/2, Windows NT.

У деяких випадках, розроблювачі відступають від цієї класики, і привілейований режим використовується й для програм ОС.

У цьому випадку, звертання програм до ядра здійснюються швидше, але при цьому відсутній надійний апаратний захист пам'яті.

За своєю внутрішньою архітектурою ОС можна умовно поділити на монолітні ОС, ОС на основі мікроядра та об’єктно-орієнтовані ОС. ОС з монолітною архітектурою можна представити у вигляді:

- прикладний рівень – прикладні процеси;
- системний рівень – монолітне ядро ОС , що, у свою чергу, складається:

  - з інтерфейсу між додатками та ядром;

  - власне ядра ОС;

- інтерфейсу між драйверами та пристроями.

Основною перевагою монолітної архітектури є більша швидкодія, у порівнянні з іншими архітектурами. Однак, ця перевага досягається, в основному, за рахунок написання основної частини програми ОС на асемблері.

Недоліки монолітної архітектури:

- системні виклики, що потребують переключення рівня привілеїв, повинні реалізовувати інтерфейси між додатками та ядром як переривання. Це значно збільшує час їх виконання;

- ядро не може бути перерване додатком. Це може призвести до того, що високопріоритетна задача може не отримати процесор, зайнятий виконанням низькопріоритетної задачі. Наприклад, низькопріоритетна задача запросила виділити пам'ять, зробила системний виклик, до закінчення якого сигнал на активізацію високопріоритетної задачі не може її активізувати;

- складність переходу на нову архітектуру процесора при невідповідності асемблерів;

- негнучкість та складнощі з розвитком, оскільки зміна частини ядра потребує його повної перекомпіляції.

ОС на основі модульної організації прибирає недоліки, пов'язані з інтерфейсом між додатками та ядром, полегшує модернізацію ОС та її встановлення на нові мікропроцесори.

При модульній архітектурі інтерфейс відіграє тільки одну роль – забезпечує зв'язки додатків із спеціальним модулем – менеджером процесів. А ядро – подвійну роль:

- керує взаємодією частин системи (менеджерів процесів та файлів);

- забезпечує неперервність виконання коду системи (тобто переключення задач відсутнє під час виконання функцій ядра).

В модульній архітектурі недоліки майже залишились, хоча вони перейшли з рівня інтерфейсу на рівень ядра.

1.3 Багатошарова структура ОС

Обчислювальну систему, що працює під керуванням ОС на основі ядра, можна розглядати як систему, що складається із трьох ієрархічно розташованих шарів:

- нижній шар – утворюється апаратурою;
- проміжний – ядро;

- верхній шар – модулі, що реалізують допоміжні функції.

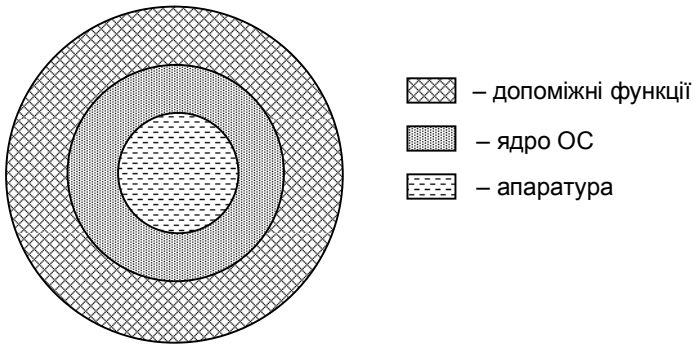


Рис. 1.2. Трьохшарова схема обчислювальної системи

Багатошарову структуру, зображену на рис. 1.2, зручно представляти для ілюстрації того факту, що кожен шар може взаємодіяти тільки із сусіднім шаром. При такій організації додатки не можуть безпосередньо взаємодіяти з апаратурою, а тільки через шар ядра.

Багатошаровий підхід є універсальним й ефективним способом декомпозиції складних систем. Відповідно до нього, система складається з ієрархії шарів. Кожен шар обслуговує вищележачий шар, виконуючи для нього деякий набір функцій, які утворюють міжшаровий інтерфейс.

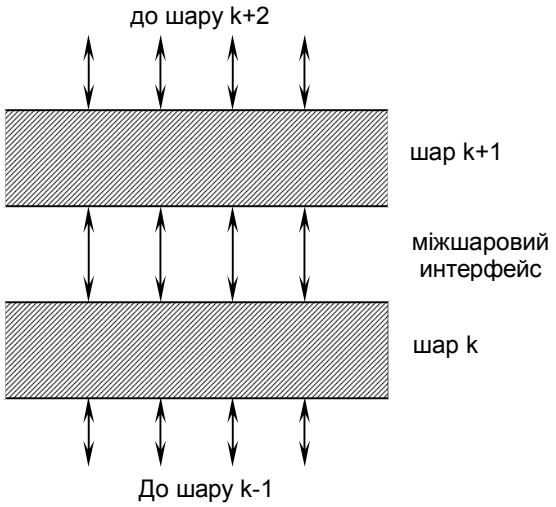


Рис. 1.3. Концепція багатошарової взаємодії

На основі функцій лежачого нижче шару, наступний шар будує свої функції – більш складні й могутніші, які, у свою чергу, стають примітивами для створення ще більш потужних функцій лежачого вище шару.

Строгі правила взаємодії обумовлюються тільки між шарами. Усередині шару зв'язки між модулями можуть бути довільними. Окремий модуль може виконати свою роботу як самостійно, так і звернутися до іншого модуля свого шару, або до лежачого нижче шару через міжшаровий інтерфейс.

Достоїнства такої системи організації:

- спрощується розробка системи;
- при модернізації системи легко робити зміни усередині кожного шару, не турбуючись про інші шари.

Оскільки ядро являє собою складний багатофункціональний комплекс, цей підхід поширюють і на структуру ядра.

Наприклад:

- **Засоби апаратної підтримки ОС.** Сюди відносять не всі апаратні засоби, а тільки ті, що прямо беруть участь в організації обчислювальних процесів: засоби підтримки привілейованого режиму, систему переривань, засоби перемикання контекстів процесів, засоби захисту областей пам'яті.
- **Машинно-залежні компоненти ОС.** Шар утворюють програмні модулі, які відображують специфіку апаратної платформи комп'ютера. Ідеально, цей шар повністю екранує лежачі вище шари ядра від особливостей апаратури. Це дозволяє розробляти ці шари на основі машинно-незалежних модулів для всіх типів апаратних платформ, підтримуваних даною ОС.
- **Базові механізми ядра**. Шар виконує найбільш примітивні операції ядра, такі як перемикання контекстів процесів, диспетчеризацію переривань, переміщення сторінок пам'яті на диск і зворотньо і т.п. Модулі даного шару не ухвалюють рішення щодо розподілу ресурсів, вони є виконавчими механізмами для модулів верхніх шарів.
- **Менеджери ресурсів.** Шар складається з потужних функціональних модулів, що реалізують стратегічні завдання по керуванню

основними ресурсами обчислювальної системи. Тут працюють менеджери (диспетчери) процесів, введення-виведення, файлової системи й оперативної пам'яті. Розбивка на менеджери може бути різною. Наприклад, менеджер файлової системи іноді поєднують із менеджером введення-виведення, а функції керування доступом користувачів до системи в цілому і її окремим об'єктам доручають окремому менеджеру безпеки.

Кожний з менеджерів веде облік вільних і використовуваних ресурсів певного типу й планує їхній розподіл відповідно до запитів програм. Для виконання ухвалених рішень менеджер звертається до нижчележачого шару. Усередині шару менеджерів існують тісні взаємозв'язки, оскільки для виконання процесу потрібен доступ одночасно до декількох ресурсів – наприклад, процесору, пам'яті, введення-виведення тощо.

- **Інтерфейс системних викликів**. Це самий верхній шар ядра. Він взаємодіє безпосередньо з додатками й системними утилітами, утворюючи прикладний програмний інтерфейс ОС. Функції, які обслуговують системні виклики, надають доступ до ресурсів системи в зручній і компактній формі, без деталей їхнього фізичного розташування. Для здійснення таких дій системні виклики звичайно звертаються по допомогу до функцій шару менеджерів ресурсів.

Надане розбиття ядра ОС на шари є досить умовною. Таке розбиття й розподіл функцій може бути іншим.

Також може бути іншим і спосіб взаємодії шарів. Для прискорення роботи ядра, в ряді випадків, відбувається звернення з верхнього шару до функцій шарів нижнього рівня, минаючи проміжні шари.

Вибір кількості шарів ядра є відповідальною й складною справою: збільшення числа шарів веде до вповільнення роботи ядра, а зменшення – погіршує розширюваність і логічність системи.

Незалежно від того, які функції виконує ОС, вона повинна задовольняти експлуатаційним вимогам. Вона, зокрема, повинна мати наступні якості:

1. **Надійність.** У випадку помилки в програмному або апаратному устаткуванні, система повинна виявити помилку й / або спробувати виправити ситуацію. Або сповістити про це користувачеві й намагатися звести збитки до мінімуму.
2. **Захист.** Користувач не хоче, щоб інші користувачі (якщо він, наприклад, працює в мережі) йому заважали. Тому ОС повинна захищати користувача від впливу чужих помилок і від спроб злочинного втручання.
3. **Ефективність.** ОС – досить складна програма, що використовує значну частину ресурсів для своїх власних потреб. Ресурси, які використовує ОС, не надходять у розпорядження користувача. Отже ОС повинна бути, якомога найбільш ощадливою. Крім того, вона повинна управляти ресурсами користувачів так, щоб звести до мінімуму час затримки й простоїв.
4. **Передбачуваність.** Вимоги, які користувач може пред'являти до системи, у більшості випадків, є непередбачуваними. Але користувач бажає, щоб обслуговування не дуже сильно змінювалося протягом тривалого часу. Зокрема, при введенні програми в машину користувач повинен мати засноване на попередньому досвіді приблизне уявлення про те, коли йому варто очікувати на результати.
5. **Зручність.** Все ясно, як і те, що універсальних зручностей не існує. Тут мова може йти про певний клас завдань.

Дещо про виконувані функції:

Розподіл процесора. У випадку “нерозумної” системи, вся вона розподіляється як єдиний ресурс. Користувач або розпоряджається машиною, або чекає, коли вона буде надана в його розпорядження. Таку стратегію дуже легко організувати, але вона не буде ефективно використовувати

устаткування. Для забезпечення паралельної з процесором роботи можна зробити так, щоб одна програма виконувала операції вводу- виводу, поки інша займає головний процесор. Хоча реалізація такого підходу є складнішою, маємо перевагу – кожен пристрій використовується інтенсивніше.

Керування пам'яттю. Керування пам'яттю тісно пов'язано з розподілом процесора. Програми можуть працювати тільки тоді, коли вони перебувають в оперативній пам'яті, але не обов'язково їх тримати там, якщо шанс отримати процесор є незначним. У цьому випадку, виявиться, що займана ними пам'ять пропадає даремно.

Оперативна пам'ять – це теж розподілюваний ресурс. Тому система витрачає час для раціонального розташування інформації, намагаючись тримати корисні програми в оперативній пам'яті й знищувати “вільні проміжки” між програмами. Для цього система може використовувати переміщення програм. Це робиться для зменшення обсягу даремно використовуваної пам'яті. Переміщення легше здійснити, якщо використовувати спеціальні стратегії організації пам'яті. Вони дозволяють ОС досить гнучко регулювати обмін інформацією між оперативною й допоміжною пам'яттю.

Зовнішні пристрої. Методи розподілу пристроїв введення-виведення й каналів зв'язку різні. Завдання використовує периферійний пристрій стільки часу, скільки йому потрібно, а потім відмовляється від нього. Пристрій зі швидким довільним доступом, такий як, накопичувач на дисках, можна спільно використовувати в декількох завданнях за принципом “операція за операцією”, тобто декільком завданням дозволяється використовувати пристрій почергово. Якщо, скажемо, два завдання спробують виконати операцію введення-виведення на одному пристрої одночасно, тоді виникає черга й затримка.

Підхід до розподілу пристроїв впливає як на коректність, так і на ефективність роботи.

Стратегія розподілу впливає й на ефективність використання пристроїв, наприклад, того ж диску.

Ефективний розподіл периферійних пристроїв важко реалізувати з двох причин:

По-перше. за допомогою існуючих математичних методів не можливо провести необхідні дослідження й відшукати оптимальні способи розподілу декількох різних типів пристроїв для загального випадку рішення завдань.

По-друге, ефективність стратегії розподілу дуже важко виміряти.

Можна оцінити тільки загальний зовнішній прояв неправильного розподілу. Якщо ж при оцінці враховувати й вплив взаємодії з оперативною пам'яттю й центральним процесором, то аналітичні й емпіричні методи виявляються ще менш перспективними. Методи розподілу пристроїв введення-виведення, контролерів і каналів сильно залежать від пристроїв.

Програмні ресурси. Часто в операційних системах є системи прикладних програм і бібліотеки програм користувачів. Будучи ресурсами, що підлягають розподілу, ці бібліотеки мають багато спільного з апаратними ресурсами.

Спільне використання інтерпретаторів, редакторів текстів та ін. можна організувати, якщо ці програми допускають паралельне використання.

Якщо в них робоча частина повністю відділена від даних й операцій запису в пам’ять і застосовується тільки до розділу даних, то такі програми допускають паралельне використання. Якщо ж програма не допускає паралельного використання, то кожен її екземпляр може бути в цей момент надано тільки одному користувачеві, так само, як і будь-який апаратний пристрій.

При паралельному використанні кожному користувачеві виділяється особистий екземпляр розділу даних, а з єдиним екземпляром робочої частини всі користувачі можуть працювати в режимі поділу.

1.4 Супервізор

Традиційний підхід при проектуванні ОС полягає в тому, що безліч процесів, що виконують основні функції системи, підпорядковуються головній програмі, яку називають супервізором. Здійснюючи централізоване керування, супервізор зв'язує воєдино інші частини системи. Він організує спільну роботу програм, установлюючи привілеї або призначаючи покарання. Він забезпечує засоби зв'язку й синхронізації між процесами й фізичними пристроями. Звичайно й повідомлення, що передані від процесу до процесу, і запуск і закінчення роботи пристроїв, і сигнали від устаткування обробляються супервізором.

Супервізор, звичайно, управляє поділом всіх ресурсів і послуг системи між користувачами.

Функції супервізора. Функції супервізора можна загалом розділити на чотири частини:

- контроль і керування;
- організація зв'язків;
- захист й обмеження;
- обслуговуючі програми.

Контроль і керування. В обов'язки супервізора входить установлення послідовності й контроль керування завданнями у системі. Для обробки завдань використовується планування порядку виконання завдань, облік споживання ресурсів й інтерпретація мови керування завданнями. Ці функції можна реалізувати незалежно від супервізора або підпорядкувати йому.

Користувач описує свої вимоги на деякій мові керування завданнями. Супервізор інтерпретує ці вимоги й повідомляє різним програмам, що розподіляють ресурси, на які саме ресурси й завдання є запити. Розподільники ресурсів обслуговують запити, можливо, у порядку їх пріоритетності. Програма обліку запам'ятовує кількість спожитих ресурсів.

Організація зв'язку. У супервізорі передбачені засоби зв'язку між різними програмами. Коли дві незалежні програми такі, як файлова система й програми керування пам'яттю, хочуть зв'язатися один з одним, вони повинні просити супервізор установити контакт. Після встановлення первісного контакту, програмам дозволяється передавати один одному повідомлення, прийнятим у даній системі способом, скажімо, за допомогою "листів" і загальної "поштової скриньки". Первісний зв'язок встановлюється через супервізор. Доступна тільки супервізору інформація дозволяє йому також встановлювати контакти між системними програмами й програмами користувачів.

Захист й обмеження. Для забезпечення гарантій виконання роботи супервізор повинен накласти деякі обмеження, як на систему, так і на користувачів. Є ряд програм, що оброблюють сигнали від апаратури, особливо відхилення від нормальних умов функціонування. Наприклад, під час роботи програми копіювання може виникнути кілька десятків різних особливих ситуацій, наприклад, збій при читанні або запису, неготовність дисководів до читання або запису, відсутність місця на дискеті для копійованого файлу тощо.

Коли надходить сигнал про помилку (звичайно це переривання), супервізор повинен прийняти рішення про повторення або навіть відхилення завдання. Для всіх цих ситуацій необхідно передбачити відповідні повідомлення й коригувальні дії.

Обмеження повинні накладати на час роботи програми й кількість видаваних нею результатів.

Супервізор може також організувати систему захисту за допомогою спеціальних "паролів".

Доступ до захищених ресурсів дозволяється тільки за відповідним паролем.

З метою захисту супервізора, в деяких системах існують два режими роботи:

- режим супервізора;
- робочий режим.

Завдяки такій обережності, супервізор має особливу владу над привілейованими частинами системи й програмами користувачів. Супервізор може захищати себе, зберігаючи життєво важливу інформацію в захищеній області пам'яті, доступної тільки в режимі супервізора.

Обслуговування програми. Крім розподілу ресурсів, супервізор виконує деякі з функцій системи. Він має набір обслуговуючих програм для аварійних ситуацій, для доступу до бібліотек програм, для обробки повідомлень від програм, що працюють в режимі супервізора. Всередині область пам'яті супервізора часто перебуває область, призначена для організації зв'язків і таблиці системи захисту. У ряді випадків, коли потрібно виконати складні дії, супервізор викликає спеціальні оброблюючі програми.

Використання супервізора - це приклад побудови ОС на принципі централізованого керування.

Ідеологія централізації ключових функцій системи під контролем супервізора має свої переваги й недоліки. Одна з переваг – проста реалізація захисту. Друга перевага – простота реалізації супервізора в цілому.

З погляду розробки, легше зосередити найважливіші системні функції у підпорядкуванні супервізора, ніж розподіляти їх по всій системі. Найкраще такий підхід виправдовує себе, коли супервізор роблять відносно невеликим. На жаль, є велика спокуса передати супервізору дуже багато функцій, перетворюючи його тим самим власне в ОС.

У централізованого супервізора є й істотні недоліки. Так програмам не дозволяється встановлювати зв'язок одна з одною самостійно.

1.5 Мікроядерна архітектура

Мікроядерна архітектура – альтернатива класичному способу організації у вигляді багатошарового ядра, що працює в привілейованому режимі.

У даному випадку, в привілейованому режимі залишається працювати тільки дуже невелика частина ОС, яку звуть **мікроядром.** Мікроядро захищене від інших частин ОС і додатку (рис. 1.4).

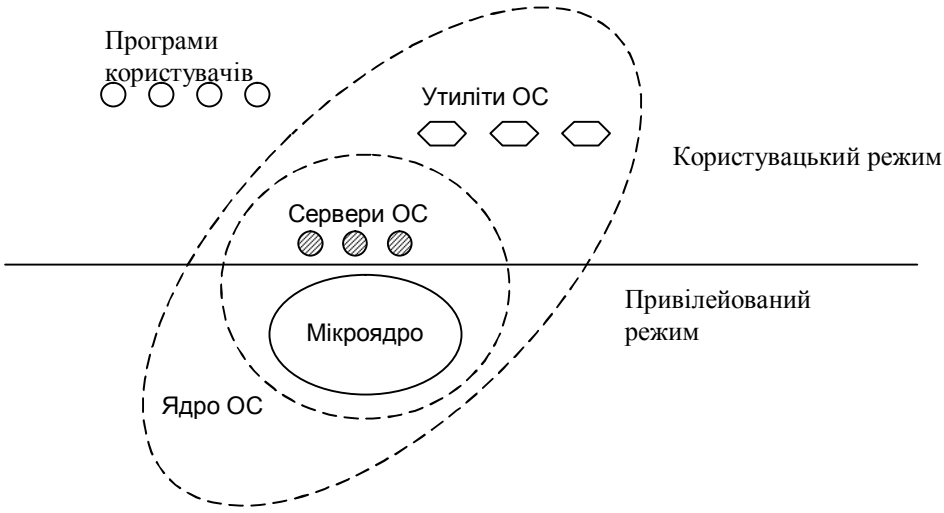


Рис. 1.4. Мікроядерна архітектура

До складу мікроядра входять:

1. Машинно-залежні модулі;
2. Модулі, що виконують базові функції ядра по керуванню процесами;
3. Обробники переривань;
4. Керування віртуальною пам'яттю;
5. Пересилання повідомлень і керування введенням-виведенням.

Інші функції ядра оформляються у вигляді програм, що працюють у користувацькому режимі. **У загальному випадку, багато менеджерів ресурсів реалізуються у вигляді модулів, що працюють у користувацькому режимі.**

Ці менеджери, однак, мають принципову відмінність від традиційних утиліт й оброблюючих програм ОС, хоча при мікроядерній архітектурі всі ці програмні компоненти також оформлені у вигляді програм.

Утиліти й оброблюючі програми викликаються, в основному, користувачами. За визначенням, основним призначенням такого додатку є

обслуговування запитів від інших програм. Тому менеджерів ресурсів, які винесені до користувацького режиму, називають **серверами ОС**.

Необхідною умовою для реалізації мікроядерної архітектури є наявність в ОС зручного й ефективного способу виклику одного процесу з іншого. Підтримка такого механізму – одне з головних завдань мікроядра.

У зв'язку з тим, що при створенні додатків широко використовується об'єктно-орієнтований підхід, поєднання об'єктно-орієнтованих додатків з процедурами ОС, особливо реального часу, призводить до побудови ОС РЧ на базі об'єктно-орієнтованого підходу.

1.6 Об'єктна архітектура на основі мікроядер

В основі цієї архітектури інтерфейс між додатками та ядром не використовується взагалі. Взаємодія між компонентами системи (мікроядрами) та користувацькими програмами виконується шляхом звичайного виклику функцій, оскільки система і додатки написані однією мові програмування (наприклад, C++). Це забезпечує максимальну швидкість системних викликів.

Фактична рівноправність усіх компонентів системи забезпечує можливість переключення задач у будь-який час.

Об'єктно-орієнтований підхід забезпечує модульність, безпеку, можливість модернізації та повторного використання коду.

Роль інтерфейсу відіграє компілятор і динамічний редактор об'єктних зв'язків (linker). При старті додатку динамічний linker завантажує необхідні йому мікроядра (на відміну від попередньо розглянутого варіанту, де попередньо завантажено усі компоненти системи). Ящо деякі компоненти мікроядра попередньо вже були завантажені для іншого додатку, вони повторно не завантажуються, а використовується код і дані вже існуючого ядра. Це дозволяє зменшити об'єм необхідної пам'яті. Оскільки різні додатки поділяють одні мікроядро, вони мають працювати в одному адресному просторі.

Переваги мікроядерної архітектури:

- висока переносимість;
- розширюваність;
- висока надійність;
- хороші передумови для підтримки розподілених програм, оскільки використовуються механізми, аналогічні мережним: взаємодія клієнтів і серверів шляхом обміну повідомленнями.

Сервери мікроядерної ОС можуть працювати як на одному, так і на декількох процесорах.

Недоліком мікроядерної архітектури є нижча продуктивність. Це позначається на швидкості роботи прикладних середовищ, і, зрозуміло, на швидкості виконання програм.

Висновок: Мікроядерні архітектури ОС є альтернативою класичному способу побудови ОС, що передбачає виконання власних основних функцій у привілейованому режимі.

У мікроядерних ОС у привілейованому режимі залишається працювати дуже невелика частина ядра, яку називають мікроядром. Всі інші функції ядра оформлюють у вигляді програм, що працюють у користувацькому режимі.

Мікроядерні ОС задовольняють більшості вимог, запропонованих до розроблених ОС, володіють переносимістю, розширюваністю, надійністю й створюють гарні можливості для підтримки розподілених програм.

За ці якості доводиться розплачуватися зниженням продуктивності, й це є основним недоліком мікроядерної архітектури.

Мікроядерну концепцію використовують такі ОС, як Windows NT і деякі версії ОС UNIX.

1.7. Контрольні запитання до розділу 1

1. Які функції виконує операційна система?

2. За якими принципами поділяють (класифікують) операційні системи?

3. Які особливості притаманні ОС реального часу?

4. Що визначає термін «сімейство операційних систем»?

5. Що визначає термін «архітектура ОС»?

6. На які групи поділяють модулі ОС?

7. Які особливості має монолітна архітектура ОС?

8. Які існують експлуатаційні вимоги до ОС?

9. Які функції виконує супервізор ОС?

10. Які особливості організації мікроядернрї архітектури ОС?

11. Які особливості об’єктної архітектури на основі мікроядер?

12. Назвіть переваги і недоліки різних видів архітектури ОС.

Розділ 2. Процеси

2.1 Визначення процесу

Процес – основна одиниця роботи в ОС. Розглянемо спочатку найбільш знайомий тип роботи – послідовну програму.

Велику послідовну програму, написану алгоритмічною мовою, можна розділити на менші послідовні програми, що звуть процедурами. Якщо рекурсія виключена, то процедура ніколи не може приводитися в дію декількома обігами одночасно. Отже, будь-якій роботі, що виконується в цей момент, однозначно відповідає діюча процедура.

Однак, в ОС цей простий зв'язок між виконуваними роботами й процедурами втрачено. Одна процедура може працювати одночасно в декількох частинах системи. Наприклад, кілька каналів, пов'язаних з дисками, можуть розділяти одну процедуру введення й навпаки, функція, що представляє за змістом єдину операцію, наприклад “введення”, може щоразу під час своєї роботи використовувати одну або кілька різних процедур.

Взаємно однозначної відповідності між процедурами й виконуваними роботами, що існує в мовах послідовного програмування, в ОС немає. Тому поняття процедури й роботи не можна вважати еквівалентними.

Ця паралельність відображується й на програмному забезпеченні. ОС виконує безліч робіт, які виконуються майже незалежно: введення, виведення, обчислення на центральному процесорі.

Поняття процесу – формалізація ідеї “незалежної роботи”.

Периферійні пристрої й пов'язані з ними канали працюють паралельно із блоком центрального процесора. У самому центральному процесорі теж можна широко використовувати паралелізм апаратного забезпечення. У системах з декількома арифметичними процесорами й значною кількістю каналів, що працюють паралельно, організація робіт стає складним завданням.

На додаток до багатьох апаратних пристроїв, що працюють паралельно, може бути ще кілька завдань, які також виконуються паралельно.

Ці паралельно виконувані завдання взагалі не повинні враховувати існування одне одного. Коли якесь завдання віддає наказ “друкувати”, його не стосується те, що інше завдання в цей самий час здійснює запис на диск тощо.

Для отримання адекватної моделі такого функціонування, потрібно ввести одиницю роботи, яка б відображала цю паралельність, а це і є **процес**.

Оскільки роботи виконуються незалежно, і процеси, що їх представляють, працюють з різними швидкостями, повинні допускатися довільні співвідношення швидкостей виконання процесів.

Отже, **процеси** – незалежні роботи, які виконуються паралельно та з різними швидкостями.

Роботам, іноді, все ж таки необхідно обмінюватися інформацією. Тому вони не є цілком незалежними. Однак спілкування між процесами повинно проходити тільки за чіткими, чітко визначеними схемами, щоб уникнути плутанини й невизначеності.

Крім того, робиться припущення, що операції всередині процесу виконуються в чітко визначеній послідовності.

Варто спробувати дати формальне визначення поняття процесу, оскільки існує дуже багато його визначень. Нехай $X = \{x_0, x_1, ..., x_N, ...\}$ – набір змінних, що характеризують стан, який називають набором змінних станів. **Стан** описується завданням значень всіх елементів, що входять у набір змінних станів.

Простір станів для даного набору змінних станів – безліч станів, які може приймати цей набір змінних.

Дія – присвоювання значень деяким зі змінних даного набору. Послідовність станів, що належать простору станів, називають **роботою**.

Один зі способів виконати роботу полягає в послідовному застосуванні різних дій. Кожна дія породжує новий стан, що за визначенням і є робота.

Функція дії – це функція, що відображає стан у дії. Функція дії може також породжувати роботу із заданого початкового стану. Вони просто описують дію, яку треба застосувати до кожного чергового стану, а ця дія породжує новий стан і т.д. нескінченно.

На змістовному рівні, набір змінних – пам'ять, стан – вміст пам'яті, функція дії – програма.

Отже, можна визначити **процес** як трійку: простір станів, функцію дії в цьому просторі станів і особливі елементи цього простору станів – **початковий стан**.

Приклад.

Нехай задано процес **P**, який має дві змінні **x** й **y**. Роботу процесу P можна описати послідовністю станів: $\{(2,1),(4,2),(6,3),(8,4)...(2i,i)...\}$.

Роботу процесу можна також описати, вказавши простір станів {(i, j), де i, j – натуральні числа}, один з початкових станів $(2,1)$ і функцію дії, що виконує дії над всіма станами: $(x,y)\to(x+2,y+1)$.

Кожен стан процесу – це "миттєвий знімок" ходу роботи, що виконує процес. Нехай програма разом зі своїми змінними перебуває в оперативній пам'яті. Можна слідкувати за виконанням програми, спостерігаючи за пов'язаними з нею комірками й регістрами. Послідовність станів описує хід роботи програми в даному середовищі.

Очевидно, що при наявності окремих наборів змінних станів у двох процесів, вони не можуть взаємодіяти. Спілкування між процесами забезпечується поділюваними змінними.

У ряді ОС, крім поняття «процес», виділяється дрібніша одиниця, що має назву **потік**. У чому ж різниця?

В ОС, де існують і процеси, і потоки, процес розглядається, як заявка на споживання всіх видів ресурсів, крім одного – процесорного часу.

Це найважливіший ресурс розподіляється ОС між потоками, які одержали свою назву завдяки тому, що вони являють собою послідовності

(потоки) виконання команд. **Потоки** виникли як засіб розпаралелювання обчислень.

У найпростішому випадку, процес складається з одного потоку. Надалі будемо, з метою спрощення, ототожнювати ці поняття й користуватися терміном **процес.**

2.2 Реалізація процесів

Кожен програмний процес однозначно визначається інформаційною структурою, яку називають **дескриптором процесу.** У типовій системі дескриптор процесу складається з:

1. Змінної стану, що визначає положення процесу (готовий до роботи, працюючий, заблокований).
2. Захищеної області пам'яті, у якій перебувають поточні значення регістрів, коли процес переривається, не закінчивши роботи.
3. Інформації про ресурси, якими процес володіє або має право користуватися.

Крім цього, у дескрипторі процесу може бути відведене місце для організації спілкування з іншими процесам.

Дуже важливо розрізняти абстрактний і програмний процеси.

Програмний процес – це й абстрактний процес, але зворотно не завжди вірно.

Дескриптор й область пам'яті, з якої складається програмний процес, повинні бути виділені з наявних у машині ресурсів.

Є два підходи до створення програмних процесів.

Можна побудувати систему з фіксованою кількістю програмних процесів, які існують завжди. У такому випадку, програмні процеси буде створено одночасно із системою.

Щоб виконати роботу, необхідно тільки одержати у своє розпорядження один з існуючих програмних процесів.

Абстрактних процесів може бути більше, ніж є програмних процесів. Тому абстрактному процесу, можливо, прийдеться очікувати, коли йому буде наданий програмний процес для виконання роботи.

Інший шлях полягає в тому, що в системі передбачають механізм для створення і знищення програмних процесів при надходженні відповідного запиту.

Цей механізм, який сам не є програмним процесом, дає системі можливість маніпулювати програмними процесами. Його називають **стрижнем**.

У процесі може викликатися створення або знищення інших процесів. Так один процес може зажадати, щоб стрижень утворив інший процес. Стрижень створює дескриптор нового процесу, виділяє для процесів пам'ять і повідомляє про завершення цієї діяльності утворюючому процесу. Тоді цей процес поміщає в пам'ять нового процесу програму (тобто подання деякої функції дії). У стрижні передбачена команда “запустити”, що виділяє процесу процесор.

Аналогічно команда “зупинити” відбирає процесор у процесу, а команда “знищити” відбирає дескриптор і ресурси.

У системі з єдиним процесором може існувати кілька процесів. При цьому стрижень дає кожному процесу можливість користуватися процесором у певні моменти часу.

Процеси можуть існувати як не зв'язані одна з одною одиниці, або можуть бути зв'язані особливими відносинами, утворюючи структури. Якщо система не передбачає такої структурної організації, тоді супервізор зобов'язаний, якщо потрібно, створювати процеси. З появою завдання супервізор створить процес для виконання завдання. Коли завдання виконано, процес знищується. При такій організації всі процеси рівні.

У більш складних системах процеси можуть бути нерівними. Наприклад, вони можуть становити деревоподібну структуру.

У системі з деревоподібною структурою процес називають **батьком** всіх процесів, які він створить, і процес називають **сином** того процесу, що його створив. Тут має місце визначення предків і нащадків. **Генеалогічне дерево** процесів у системі являє собою орієнтований граф, де кожен процес представляється вершиною, а дуга виходить із вершини **A** і заходить у вершину **B** тоді й тільки тоді, коли **A** – батько **B**. Генеалогічне дерево описує впорядкованість процесів усередині системи в будь-який момент часу.

У системі з деревоподібною структурою (рис. 2.1) можуть бути дуже чіткі правила щодо передачі ресурсів й організації керування.

Наприклад, при створенні кожен процес може одержати лише ті ресурси, які “належать” його батькові. Батько може мати право контролювати дії своїх синів і приймати дії для виправлення ненормальних ситуацій.

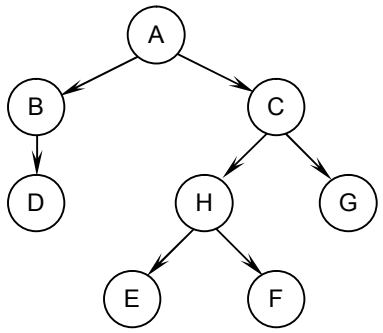


Рис. 2.1. Дерево створення процесів

Така структурована система має переваги:

- Розподіл ресурсів перебуває під строгим контролем. Ресурси всякого процесу колись були власністю кожного з його предків. Якщо процес хоче звільнити деякі зі своїх ресурсів, він може віддати їх тільки своєму батькові. Ніколи не буває незрозумілим, який ресурс належить якому з процесів.
- Вся структура, в цілому, така, що одні з процесів мають більше прав, ніж інші, тобто в ній передбачено простий механізм поділу роботи.

- Завдяки структурі завжди зрозуміло, якому процесу належить керування: батько управляє синами.

У безструктурній системі все знає й виконує супервізор. Хоча централізоване керування допускає вільне розбиття завдань, це ускладнює ведення обліку, оскільки супервізору доводиться стежити за кожним процесом у системі.

2.3.Спілкування між процесами

Для ефективного використання апаратних й програмних ресурсів їх поділяють.

Розглянемо завдання користувача, що виконується в системі, як єдиний програмний процес. Рідко буває, коли процесу протягом усього часу його існування потрібен деякий апаратний пристрій для одноосібного користування.

Припустимо, що він використовує канал для роботи з периферійним пристроєм. Якщо канал обслуговує вимоги процесу швидше, ніж останній їх породжує, є сенс організувати поділ єдиного каналу між декількома процесами.

Для зниження втрат через неефективне використання програмних засобів також можна скористатися поділом.

Візьмемо, наприклад, таку програму як компілятор. Якщо процесу потрібно відтранслювати програму, він може скопіювати програму компілятора у свою пам'ять і виконати компіляцію. Якщо цим компілятором користуються декілька процесів, тоді таке розмноження, з погляду ресурсів пам'яті, є неоптимальним. Доцільно розділити за часом один екземпляр компілятора.

Зауважимо, що апаратні і програмні ресурси мають загальні властивості.

Розглянемо систему керування файлами, яка у кожен момент часу може обслуговувати запити тільки одного процесу.

Якщо ця система – програмний ресурс, її поділ організується так само, як і поділ апаратних ресурсів.

Фізичні пристрої називаються **фізичними** або природними ресурсами.

Частина програмного забезпечення, що поводиться подібно фізичному ресурсу, називають **логічним ресурсом**.

Процесу неважливо, яким саме ресурсом він користується: фізичним чи логічним; важливо, щоб ресурс робив те, що повинен.

2.4 Синхронізація процесів

Незважаючи на те, що ресурси можна розділяти, вони звичайно доступні в кожен момент часу тільки одному процесу. Ресурс, що допускає обслуговування тільки одного користувача кожен раз, називають **критичним ресурсом**.

Якщо кілька процесів хочуть використовувати критичний ресурс у режимі поділу часу, їм варто синхронізувати свої дії таким чином, щоб цей ресурс завжди перебував у розпорядженні не більш, ніж одного з них.

Якщо ресурс зайнятий, то інші процеси, яким він потрібний, тимчасово одержують відмову й повинні чекати його звільнення.

Всередині кожного процесу можна виділити місця, у яких відбувається звертання до критичних ресурсів. Ці місця називають **критичними ділянками**.

Розглянемо два незалежних процеси:

```
parbegin
   Процес_1: do while (true)
                  begin
                   Критична ділянка 1;
                  end;
                 частина процесу 1, що залишилася;
              end;
   Процес_2: do while (true)
                  begin
                   Критична ділянка 2;
                  end;
                 частина процесу 2, що залишилася;
              end;
```

Конструкція вигляду

```
parbegin
  Оператор 1;
  Оператор 2;
  ...
  Оператор N;
parend.
```

означає, що оператори 1-N виконуються паралельно.

До речі, вищенаведені записи називають записами на **псевдокоді**.

Критичні ділянки процесів відповідають тим групам операцій, які звертаються до поділюваного критичного ресурсу.

Щоб не допустити одночасного виконання обох критичних ділянок, у системі повинен передбачатися механізм синхронізації цих двох процесів. Цей механізм повинен володіти двома властивостями:

1. Якщо один або кілька процесів хочуть звернутися до своїх критичних ділянок, то один з них повинен одержати дозвіл увійти у свою критичну ділянку.
2. У кожен момент часу не більш, ніж одному процесу дозволяється перебувати у своїй критичній ділянці.

Виникає не тільки проблема синхронізації процесів, але й необхідність в обміні інформацією між ними.

Позначимо словом “Постачальник” процес, що відправляє порції інформації іншому процесу, який позначимо як “Споживач”.

Порції інформації, що передаються, розглядатимо як повідомлення.

Один з методів, застосовуваних при реалізації передачі повідомлень, полягає в тому, що створюється пул вільних буферів, кожний з яких може містити одне повідомлення.

Між процесами Постачальник і Споживач є черга заповнених буферів, що містять повідомлення. Коли Постачальник хоче послати повідомлення, він додає у хвіст цієї черги ще один буфер. Споживач для одержання повідомлення забирає буфер із початку черги.

Така організація вимагає кооперування Постачальника і Споживача в багатьох областях. Вони повинні стежити за кількістю заповнених і вільних буферів. Постачальник може передавати повідомлення тільки доти, поки є вільні буфери. Споживач може одержувати повідомлення тільки, якщо черга не порожня.

Для обліку заповнених і вільних буферів потрібні поділювані змінні. Але оскільки змінні використовуються в режимі поділу, то будуть потрібні й критичні ділянки. Отже, змінювати значення лічильника заповнених буферів треба в режимі взаємного виключення.

Та ж проблема виникає при реалізації **черги повідомлень**. Нехай черга реалізована у вигляді зв'язного списку. Якщо Постачальник додає повідомлення в чергу саме в той момент, коли Споживач забирає з неї саме останнє повідомлення, може вийти неправильне посилання у списку.

Такі процеси як Постачальник і Споживач, що спілкуються на однакових правах, називають також **сопрограмами**. На жаль, у літературі є дуже багато різних визначень цього терміну. Щоб уникнути плутанини, далі будемо користуватися терміном “процеси, що кооперуються”.

Якщо процесу Постачальник важливо знати, що його повідомлення отримано, тоді Споживач повинен підтверджувати одержання кожного повідомлення, тобто Постачальник і Споживач міняються місцями.

Якщо в системі є декілька пар “Постачальник - Споживач”, можна організувати поділ вільних буферів, об'єднавши їх в загальний пул.

Зупинимося на рішенні проблем синхронізації й відносин типу “Постачальник - Споживач”.

Синхронізація за допомогою елементарних прийомів нижнього рівня

Блокування пам'яті. Взаємне виключення реалізують апаратно, зробивши операції над пам'яттю неподільними. Тобто, якщо кожний із двох процесів намагається помістити якесь значення в одну й ту ж саму комірку,

тоді суперечка вирішується апаратно: одному процесу дозволяється виконати операцію запису негайно, а іншому доводиться чекати, поки перший не закінчить. Таке рішення називається **блокуванням пам'яті**.

Повернемося до розгляду прикладу з двома паралельними процесами.

Оскільки швидкості обох процесів довільні, зазначені вище умови повинні виконуватися незалежно від співвідношення цих швидкостей.

Здавалося би, найпростіше запропонувати наступне рішення:

```
Перемикач_1, Перемикач_2:boolean;
Begin
 Перемикач_1:=False;
 Перемикач_2:=False;
 parbegin
  Процес_1: do while (True);
     Цикл_1:     do while (Перемикач_2);
                 end;
                Перемикач_1:=True;
                /* Критична ділянка 1 */
                Перемикач_1:=False;
                /* частина процесу 1, що залишилася, */
             end;
  Процес_2: do while (True);
     Цикл_2:     do while (Перемикач_1);
                 end;
                Перемикач_2:=True;
                /* Критична ділянка 2 */
                Перемикач_2:=False;
                /* частина процесу 2, що залишилася, */
             end;
 parend;
End.
```

Оскільки швидкості довільні, припустимо, що Процес 2 працює набагато швидше, ніж Процес 1. Настільки швидше, що фактично після виявлення Процесом 1, що в Перемикачі 2 записано ”`False` ”, але перш ніж він встигне встановити значення “`True`” у Перемикачі 1, Процес 2 пробігає свою частину, що залишилася, і перескакує через Цикл 2 (оскільки в Перемикачі 1 все ще значення “`False`”). У такому випадку, обидва процеси перейдуть до виконання (одночасно) своїх критичних ділянок, тобто в такому перемикальному випадку система буде працювати неправильно.

Пропонуємо самостійно спробувати змінити наведену програму так, щоб вона задовольняла обмеженням, які накладаються на рішення проблеми критичної ділянки.

Алгоритм Деккера

Приведемо варіант рішення, запропонованого Деккером.

```
C1,C2,Черга:integer;
begin
 C1:=0;
 C2:=0;
 Черга:=1;
 parbegin
  Процес_1: begin C1:=1;
                  do while (C2=1);
                   if Черга=2 then
                      begin
                       C1:=0;
                        do while (Черга=2);
                        end;
                       C1:=1;
                      end;
                  end;
                  Критична ділянка Процесу 1;
                  C1:=0;
                  Черга:=2;
                  частина процесу 1, що залишилася,;
             end;
   Процес_2: begin C2:=1;
                  do while (C1=1);
                   if Черга=1 then
                      begin
                       C2:=0;
                        do while (Черга=1);
                        end;
                       C2:=1;
                      end;
                  end;
                  Критична ділянка Процесу 2;
                  C2:=0;
                  Черга:=1;
                  частина Процесу 2, що залишилася,;
             end;
 parend;
end.
```

Тут введена додаткова змінна Черга, що визначає, чия саме черга спробувати увійти, за умови, що обидва процеси хочуть виконати свої

критичні ділянки. Це рішення має узагальнення для випадку довільного числа процесів, що конкурують через критичний ресурс.

Перевірка та установка. Ця машинна операція значно спрощує рішення проблеми критичної ділянки.

До операції “Перевірка й Установка” звертаються з двома параметрами: “Локальний” (Local) й “Загальний” (Common). Операція бере значення параметра “Загальний” і присвоює його змінній “Локальний”, а потім встановлює змінній “Загальний” значення 1. Ця операція э однією з апаратних засобів вирішення задачі критичного інтервалу. Вона уперше була реалізована у IBM-360. Там вона дістала назву TS (Test and Set). Команда TS двоадресна. Її дія полягає у присвоєнні процесором першому операнду значення другого операнда, після чого другому операнду присвоюється значення 1. Ця змінна Common TS буде загальною для усіх процесів. Вона буде дорівнювати 1, коли один із процесів знаходиться в критичній області. Якщо процес хоче увійти в критичну область, його локальна змінна приймає значення 1.

Головна властивість цієї операції – її неподільність. Коли процес виконує операцію “Перевірка й Установка” ніяких інших дій не може відбутися між її початком і закінченням.

Змінна “Загальний” розділяється між процесами, що підлягають синхронізації стосовно деякого критичного ресурсу. У кожного процесу є своя власна змінна “Локальний”.

Якщо “Загальний” = 1, це значить, що якийсь процес перебуває у своїй критичній ділянці. Початкове значення змінної “Загальний” = 0.

Надамо рішення проблеми методом “Перевірка й Установка”. У цьому рішенні припускається, що в машині передбачено блокування пам'яті**, тобто операція “Загальний” := 0 є неподільною**.

Алгоритм перевірки й установки

```
Var Common, Local1, Local2: integer;
begin
     Common:= 0;
parbegin
     ПР1: while true do
                begin
                    Local1:= 1;
                    while Local1=1 do TS(Local1, Common);
                 <критична ділянка ПР1>;
                    Common:=0;
                 <залишок ПР1>;
                 end;
                 end;
     ПР2: while true do
                 begin
                 Local2:=1;
                 while Local2=1 do TS(Local2, Common);
                 <критична ділянка ПР2>;
                    common:=0;
                 <залишок ПР2>;
       end
       end;
  parend.
```

Нехай першим хоче увійти у свою критичну ділянку ПР1. У цьому випадку, Local1 спочатку прийме значення 1, а після циклу перевірки за допомогою команди TS(Local1, common) дорівнюватиме 0. При цьому значення Common дорівнюватиме 1. ПР1 увійде у критичну ділянку. Після виходу з критичної ділянки Common=0, що дасть можливість ПР2 увійти у свою критичну ділянку.

При цьому вважаємо, що операція Common:=0 – неподільна завдяки операції *блокування памяті.*

Недолік цього методу полягає у тому, що у циклі перевірки значення змінної Common процесорний час непродуктивно використовується. Ситуація ще погіршується, якщо під час виконання ПР1 у його критичній секції виникне переривання, а ПР1 почне виконання. Він увійде у цикл перевірки і не вийде з нього доти, доки ПР1 не вийде з переривання.

До речі, у мікропроцесорів Intel, починаючи з 80/386 існують спеціальні команди під назвами BTC (Bit Test and Reset), BTS, BTR, які є прототипами команди “Перевірка та Установка.”

Обидва розглянутих метода синхронізації можуть виявитися досить неефективними, оскільки щоразу, коли один процес виконує свою критичну ділянку, будь-який інший процес, що теж хоче увійти в критичну ділянку, попадає в цикл і повинен там очікувати дозволу.

При такому очікуванні в циклі, що називають **активним очікуванням**, даремно витрачається час центрального процесора.

Семафори. Одним із методів, що дозволяють уникнути активного очікування, є метод використання **семафорів.**

Семафор – ціла змінна, значення якої можуть змінювати тільки операції **P** й **V**. Нехай **S** – семафор. Коли процес виконує операцію **P(S)**, **S** зменшується на одиницю, і далі:

1. Якщо **S≥0**, то процес продовжує роботу.
2. Якщо **S<0**, то процес зупиняється й стає в чергу очікування, пов'язану з **S**. Він залишається заблокованим доти, поки операція **V(S)**, виконана іншим процесом, не звільнить його.

Коли процес виконує **V(S), S** збільшується на одиницю та

1. Якщо **S>0**, процес продовжує роботу.
2. Якщо **S≤0**, то один процес вилучається із черги очікування й одержує дозвіл продовжити роботу. Процес, що звернувся до операції **V(S)**, теж може продовжувати роботу.

Крім того, операції **P** й **V** – неподільні. У кожен момент часу тільки один процес може виконувати операцію **P** або **V** над даним семафором. Тому, якщо **S=1** і два процеси одночасно спробують виконати операцію **P(S)**, то тільки одному з них буде дозволено продовжити роботу. Інший процес буде заблоковано і поставлено в чергу до семафора **S**.

Раніше вказувалось, що стрижень системи – це механізм, що реалізує процеси. Стрижень повинен виділяти процесам процесор і відбирати в них його. Операції **P** й **V** можна реалізувати всередині стрижня. Таким чином забезпечується неподільність цієї операції.

Семафор, максимальне значення якого дорівнює одиниці, називають **двійковим семафором**.

За допомогою двійкового семафора процеси можуть організувати взаємне виключення, використовуючи операції **P(S)** і **V(S).**

Приклад.

```
Var ВІЛЬНИЙ:integer;
begin
 ВІЛЬНИЙ:=1;
 parbegin
     Процес_1: begin  do while (True);
                    <Початок Процесу 1>;
                    P(ВІЛЬНИЙ);
                    <Критична ділянка 1>;
                    V(ВІЛЬНИЙ);
                    <частина Процесу 1, що залишилася >;
                    end;
               end;
     Процес_2: begin  do while (True);
                    <Початок Процесу 2>;
                    P(ВІЛЬНИЙ);
                    <Критична ділянка 2>;
                    V(ВІЛЬНИЙ);
                    <частина Процесу 2, що залишилася >;
                     end;
               end;
 parend;
end.
```

Тут **S** приймає значення 1, 0, -1. Якщо **S=1**, це значить, що жоден процес не перебуває у своїй критичній ділянці. Якщо **S=0 –** один процес перебуває у своїй критичній ділянці. Якщо **S=-1 –** один процес перебуває у своїй критичній ділянці, а другий – у черзі очікування.

Рішення, що використовують двійкові семафори, застосовуються так само й при великій кількості процесів. Якщо алгоритм Деккера стає дуже складним для більш ніж двох процесів, то рішення з семафором залишається тривіальним. У цьому головна перевага семафора.

Яким чином можна реалізувати виконання семафорних операцій? Наприклад:

```
P(S):   if S >=1 then S:= S -1
        else wait(S); {зупинити процес та помістити його у
        чергу очікування до семафора S}
V(S):    if S <0 then Release(S); {помістити один з очікуючих
        процесів, що знаходяться у черзі до семафора S, у чергу
        готових до виконання процесів}
         S:= S +1.
```

Wait означає, що супервізор ОС повинен перевести задачу у стан очікування, тобто відправити її у чергу семафора S.

Виклик Release означає звернення до диспетчера задач з проханням перевести перший з процесів, що стоять у черзі до семафора S, у стан готовності.

Розглянемо приклад. Однопроцесорна система. У ній неподільність операцій P(S) та V(S) можна забезпечити блокуванням переривань. Семафор S можна реалізувати у вигляді запису з двома полями. В одному – значення змінної S, у другому – вказівник на список процесів, що заблоковані на семафорі S.

```
Type semafore= record
     лічильник: integer;
     вказівник: pointer
     end;
var S:semaphore;

procedure P(var S: semaphore);
     begin
          <заборонити переривання>;
          лічильник:= S.лічильник-1;
          if S.лічильник <0 then wait(S);
          {вставити процес, що звернувся у список
          S.вказівник, та надати процесор процесу з черги на
          виконання}
          <дозволити переривання >
     end;

procedureV(var S: semaphore);
     begin
          <заблокувати переривання>;
          S.лічильник:= S.лічильник+1:
          if S.лічильник<=0 then
          Release(S);
          {Разблокувати перший процес із черги S.вказівник}
```

```
            <дозволити переривання>;
        end;

  procedure InitSem(var:semaphore);
        begin
            S.лічильник:=1;
            S.вказівник:=nil;
        end;
```

Реалізація семафорів у багатопроцесорних системах більш складна. Одночасний доступ до семафора S з двох процесів блокували обмеженням переривання. Цей механізм у багатопроцесорній системі не працює. Не можна заблокувати доступ декількох процесів одночасно до семафору S за допомогою раніше використаного механізму. Тут потрібен механізм, що дозволяв би виключення доступу для декількох процесорів одночасно. Одним з рішень є використання раніше розглянутого методу “Перевірки та Установки” (подробиці можна знайти в [1]).

За допомогою загальних семафорів легко вирішується проблема “Постачальник – Споживач” з використанням поштової скриньки. Подільні змінні у цьому випадку – лічильники вільних та зайнятих буферів, які повинні бути захищені від обох процесів.

```
Var S.вільно, S.заповнено, S.взаємовиключено: semaphore;
     begin
       Init Sem(S.вільно,N);{N-кількість комірок буферу}
       Init Sem(S.заповнено,0);
       Init Sem(S.взаємовиключено,1);
parbegin
       Постачальник: while true do
            begin
               <приготувати повідомлення>;
               P(S.вільно);
               P(S.взаємовиключено);
               <передача повідомлення>;
               V(S.заповнено);
               V(S.взаємовиключено);
            end
          end;
       Споживач: while true do
            begin
               P(S.заповнено);
               P(S.взаємовиключено);
               <отримати повідомлення>;
               V(S.вільно);
```

```
                V(S.взаємовиключено>;
                <обробка повідомлення>;
                end
          end
parend
end.
```

S.вільно, S.заповнено – числові семафори. Вони використовуються як лічильники вільних та зайнятих буферів. S.взаємовиключення – двійковий семафор.

За допомогою семафорів можна організувати синхронізацію процесів, коли завершення одного процесу пов'язано з очікуванням завершення іншого.

Нехай існують два процеси: П1 та П2. Необхідно, щоб П1 запускав процес П2 та чекав його виконаня, тобто П1 не продовжував би своє виконання до тих пір, поки процес П2 до кінця не виконав би свою роботу.

```
Var S:semaphore;
     begin
          Init Sem(S,0);
          П1: begin
               <початок П1>;
               ON(П2); {Поставити на виконання П2}
               P(S);
               <завершення П1>;
               STOP
               end;
          П2: begin
               <виконаня П2>;
               V(S);
               STOP
               end
     end.
```

Початкове значення семафору дорівнює 0 (див. процедуру Init Sem). Якщо П1 почав виконуватись першим, то через деякий час він поставить на виконання процес П2. Після цього буде виконана процедура P(S), і П1 перейде у стан очікування на семафорі S. Процес П2 виконає усі необхідні дії та виконає процедуру V(S) і відкриє семафор S. Після цього П1 знову готовий до виконання.

За допомогою загальних семафорів можна організувати керування ресурсами, наприклад, дисками, і використати їх для синхронізації процесів.

Однак, при використанні семафорів для синхронізації й керування ресурсами виникають труднощі, пов'язані з організацією черги до семафора.

Якщо значення семафора **S стає** негативним, це значить, що один або кілька процесів очікують черги, пов’язаної з S.

Коли виконується чергова операція V(S), стрижень повинен вибрати, який процес взяти із черги. Стрижень системи може обслуговувати чергу в порядку надходження або використати пріоритетну схему обслуговування. Порядок обслуговування черги повинен обумовлюватися цілями, для яких застосовується даний семафор. Питання визначення дисципліни обслуговування при проектуванні ОС тісно пов'язаний з моделюванням обчислювальних процесів, для чого можуть бути ефективно використані моделі масового обслуговування.

Пріоритет процесу можна визначати динамічно за числом дорогих ресурсів, якими цей процес розпоряджається.

Вважається, що семафори – досить ефективний засіб для задоволення потреб ОС при реалізації синхронізації й спілкування між процесами.

Проте, часто **семафори виявляються незручним засобом**. Наприклад, складно реалізувати схему **передачі повідомлень між декількома процесами**.

Елементарні прийоми синхронізації на верхньому рівні

Поштові скриньки. Якщо процес **П1** хоче спілкуватися із процесом **П2, П1** просить систему створити поштову скриньку, що зв'яже ці два процеси так, щоб вони могли передавати один одному повідомлення. Для того, щоб послати процесу **П2** якесь повідомлення, процес **П1** просто поміщає це повідомлення в поштову скриньку, звідки процес **П2** може його в будь-який час взяти. При застосуванні поштової скриньки процес **П2** зрештою обов'язково одержить повідомлення, коли звернеться за ним, якщо взагалі звернеться.

У багатьох операційних системах поштові скриньки реалізовані у вигляді логічних файлів, доступ до яких аналогічний доступу до фізичних файлів. З поштовими скриньками дозволені наступні операції: створення, відкриття, запис / читання повідомлення, закриття, видалення. У деяких системах підтримуються додаткові службові функції, наприклад лічильник повідомлень в поштовій скриньці або читання повідомлення без видалення його із скриньки.

Поштові скриньки розміщуються в оперативній пам'яті або на диску і існують лише до виключення живлення або перезавантаження. Якщо вони фізично не розташовані на диску, то вважаються тимчасовими файлами, їх знищують після виключення системи. Поштові скриньки не мають імен подібно реальним файлам – при створенні їм присвоюються логічні ідентифікатори, які використовуються процесами при зверненні.

Для створення поштової скриньки операційна система визначає покажчики на область пам'яті для операцій читання / запису і відповідні змінні для захисту доступу. Основними методами реалізації є або буфер, розмір якого задається при створенні скриньки, або зв'язаний список, який, в принципі, не накладає ніяких обмежень на кількість повідомлень в поштовій скриньці.

У найбільш поширених реалізаціях процес, який надсилає повідомлення, записує його у поштову скриньку за допомогою оператора, схожого на оператор запису у файл:

```
put_mailbox (# 1, message)
```

Аналогічно, для отримання повідомлення процес зчитує його з поштової скриньки за допомогою оператора:

```
get _mailbox (# 1, message)
```

Запис повідомлення у поштову скриньку означає, що воно просто копіюється у вказану поштову скриньку. Може трапитися, що у поштовій скриньці не вистачає місця для зберігання нового повідомлення, тобто

поштова скринька або занадто мала, або в ній зберігаються ще непрочитані повідомлення.

Поштова скринька – це інформаційна структура, для якої задаються правила, що описують її роботу. Вона складається з **головного** елемента, що містить опис даної поштової скриньки, та з декількох **гнізд**, у які поміщають повідомлення. Розмір кожного гнізда й кількість гнізд звичайно задаються при створенні поштової скриньки. Правила роботи можуть бути різними, залежно від складності поштової скриньки.

У найпростішому випадку, повідомлення передаються тільки в одному напрямку. Процес **П1** може надсилати повідомлення доти, поки є вільні гнізда. Якщо всі гнізда заповнені, то процес **П1** може або чекати, або зайнятися іншими справами й спробувати надісслати повідомлення пізніше.

Аналогічно, **П2** може одержувати повідомлення доти, поки є заповнені гнізда.

Двосторонній зв'язок. Використовується, якщо необхідно передавати підтвердження про одержання повідомлення. При цьому дозволяється передача повідомлень через поштову скриньку в обох напрямках. Якщо передавальний процес **П1** працює швидше, ніж приймаючий процес **П2, П1** може заповнити всі гнізда, не залишивши **П2** гнізд для відповідних повідомлень. Щоб цього не відбулося, вимагають, щоб відповіді надсилалися в тих самих гніздах, у яких перебували їхні повідомлення.

Багатовходові поштові скриньки. Використовують, коли декільком процесам необхідно спілкуватися з одним процесом. Прикладом може бути система керування файлами. Такі скриньки більш ефективні порівняно з тими, де кожному окремому процесу надається своя поштова скринька, проте реалізація складніша.

Порти. Для відсилання повідомлення в поштову скриньку процес повинен знати її ім'я. Іноді це незручно. **Порт** – це сполучна ланка між процесом і поштовою скринькою. Коли поштова скринька з'єднана з певним

портом, процесам для відправки повідомлення потрібно вказати тільки ім'я порту.

В одному з варіантів реалізації, у кожного із процесів може бути **вхідний** й **вихідний** порти. Поштові скриньки створюються й знищуються процесами. Коли процес створює поштову скриньку, його зв'язують з якимось портом за допомогою команди “зв'язати”. Щоб скасувати зв'язок, його замінюють зв'язком відповідно між фіктивним портом і поштовою скринькою. Власник поштової скриньки може його знищити. Пам'ять, яку вона займала, повертається процесу, що її надав.

Повернемось до семафору. Аналіз задач синхронізації показує, що очевидні переваги реалізації (простота, відсутність активного очікування) семафорних механізмів мають цілий ряд недоліків. Так, семафорні механізми не вказують безпосередньо на синхронізуючу умову, з якою він зв’язаний. Тому при побудові складних схем синхронізації алгоритми вирішення задач виявляються досить складними.. Вирішенням проблеми може стати застосування монітора.

Монітор Хоара. Монітор – набір процедур й інформаційних структур, якими процеси користуються в режимі поділу, причому в кожен момент ними може користуватися тільки один процес.

Монітор можна уявити собі як кімнату, від якої є тільки один ключ. Якщо якийсь процес має намір скористатися цією кімнатою й ключ перебуває зовні, то цей процес може відімкнути кімнату, увійти й скористатися однією із процедур монітора.

Якщо ключа зовні немає, то процесу прийдеться чекати, поки той, хто користується кімнатою в цей момент, не вийде з неї й не віддасть ключ. Крім того, у кімнаті не можна залишатися назавжди.

Розглянемо, наприклад, ресурс, котрий розподіляє деяка програма – планувальник.

Щоразу, коли процес хоче одержати у своє розпорядження якісь частини ресурсу, він повинен звернутися до планувальника. Процедура планувальника розділяє всі процеси, і кожен процес може в будь-який момент звернутися до планувальника. Але планувальник не в змозі обслуговувати одночасно більше одного процес. Виходить, що планувальник являє собою приклад монітора.

Іноді монітору необхідно затримати процес, що звернувся за наданням йому ресурсу, якщо ресурс вже кимось використовується, до вивільнення ресурсу.

Варто підкреслити, що монітор – це пасивний об'єкт, як кімната. Це не процес. Монітор оживає тільки тоді, коли який-небудь процес вирішує скористатися його послугами. **Особливість програми монітора** полягає в тому, що в будь-який момент її може виконати **тільки один процес**.

Таким чином, монітор – це механізм забезпечення паралелізму, який має як дані, так і процедури, що необхідні для реалізації динамічного розподілу загального ресурсу або групи ресурсів.

Процес, якому потрібен ресурс, що поділяється, має звернутися до монітора, який або забезпечить доступ , або відмовить.

Вхід до монітора знаходиться під жорстким контролем – виконується взаємовиключення процесів, оскільки у кожний момент часу в монітор може увійти тільки один процес, а інші процеси чекають, причому режимом очікування керує сам монітор. Він вказує умову, за якою процес має очікувати.

Внутрішні дані монітора можуть бути глобальними або локальними стосовно до однієї процедури. До всіх даних або ресурсів можна звертатись тільки із середини монітору. Процеси можуть тільки звертатись до процедур та не можуть безпосередньо звертатись до даних.

Якщо процес звертається до деякої процедури і виявляє, що відповідний ресурс зайнятий, процедура видає команду очікування Wait з певною умовою очікування. Процес, що займав ресурс, може його звільнити, однак, може

статися, що вже є процеси, що чекали звільнення цього ресурсу. Монітор виконує програму SIGNAL, за якою один з процесів, що стоять у черзі за цим ресурсом, може його отримати. Якщо деякий процес повертає ресурс, і він нікому не потрібен, то нічого не виконується, а ресурс заноситься у список вільних ресурсів.

Іноді роблять так, щоб процес, який стоїть у черзі до ресурсу, мав би вищий пріоритет ніж новий процес, що хоче оволодіти цим ресурсом.

Розглянемо приклад виділення одного ресурсу.

```
monitor Resourse;
condition free; {умова вільний}
var busy: boolean; { busy - зайнятий}
procedure Reqest; {запит}
  begin
       if busy then Wait(free);
       busy:=true;
       Take off; {видати ресурс}
       end;
Procedure Release; {звільнити}
  begin
       Take ON {отримати ресурс}
       busy:= false;
       SIGNAL(free);
end;
  begin
       busy:=false;
end.
```

Ресурс запитується та звільняється процедурами Request {запит} та Release {звільнити}. Якщо процес звернувся до процедури Request, коли ресурс використовується (busy:= true), процедура виконує команду Wait(free). Ця операція блокує не команду Request, а процес, що до неї звернувся. Коли процес, що використовує ресурс, звертається до монітора з процедурою Release, операція SIGNAL блокує процес, що стоїть першим у черзі за даним ресурсом. Він виконує процедуру Request. Якщо черги немає, ніякі дії не виконуються.

При синхронізації процесів семантика монітора гарантує, що, якщо хоча б один процес очікує виконання умови, то ніякий інший, що звернувся, не

може втрутитися між сигналом про виконання цієї умови й продовженням одного з процесів, що очікують.

Порівняно з семафорами, монітори мають наступні переваги:

- монітори – дуже гнучкий інструмент. Наприклад, за його допомогою можна реалізувати поштову скриньку;
- здійснюється локалізація всіх поділюваних змінних всередині тіла монітора, що дозволяє позбутися малоприємних конструкцій в процесах, яким потрібна синхронізація.

2.5 Контрольні запитання до розділу 2

1. Дайте визначення термінам: процес і потік.
2. В чому полягає різниця між процесом і потоком?
3. Що визначає термін дескриптор процесу? Які складові дескриптора?
4. Коли виникає необхідність у спілкуванні між процесами?
5. Коли виникає необхідність синхронізації процесів?
6. Які існують алгоритми синхронізації процесів на нижньому рівні?
7. В чому різниця між двійковим семафором та м’ютексом?
8. Які існують методи синхронізації на верхньому рівні?
9. Що таке поштова скринька і які існують способи її реалізації?
10. Яке призначення монітора і які існують методи реалізації моніторів?

Розділ 3. Тупики

Ситуація, коли процеси чекають один одного нескінченно довго, називається **тупиком** або **дедлоком (**deadlock**)**.

Існують три основні напрямки політики запобігання тупиків:

- запобігання тупиків;
- автоматичне виявлення;
- виявлення за участю оператора.

Метод автоматичного виявлення тупиків допускає потрапляння системи в тупикову ситуацію, проте з можливістю виявити це програмним шляхом. Потім система відбирає ресурси в інших процесів і віддає їх на те, щоб зрушити з місця процеси, які потрапили в тупик.

Третій підхід базується на тому, що тупикові ситуації виникають занадто рідко, щоб про них слід було турбуватися. Коли така ситуація все ж таки виникає, оператор її виявляє й перезапускає систему. Іноді це обходиться занадто дорого, зокрема, дискредитує систему в очах користувача.

3.1 Алгоритм запобігання тупикових ситуацій

Розглянемо систему, в якій процеси конкурують через стрічкопротягувальні пристрої. Якщо процесу надати всі необхідні йому пристрої, він буде працювати протягом кінцевого відрізка часу, після чого повертає всі надані йому пристрої.

Розглянемо приклад. Нехай двом процесам, що виконуються в режимі мультипрограмування, для виконання роботи потрібно два ресурси, наприклад, принтер і послідовний порт. Така ситуація може виникнути, наприклад, під час роздруківки інформації, що надходить модемним зв'язком.

Представимо фрагмент двох програм (рис. 3.1):

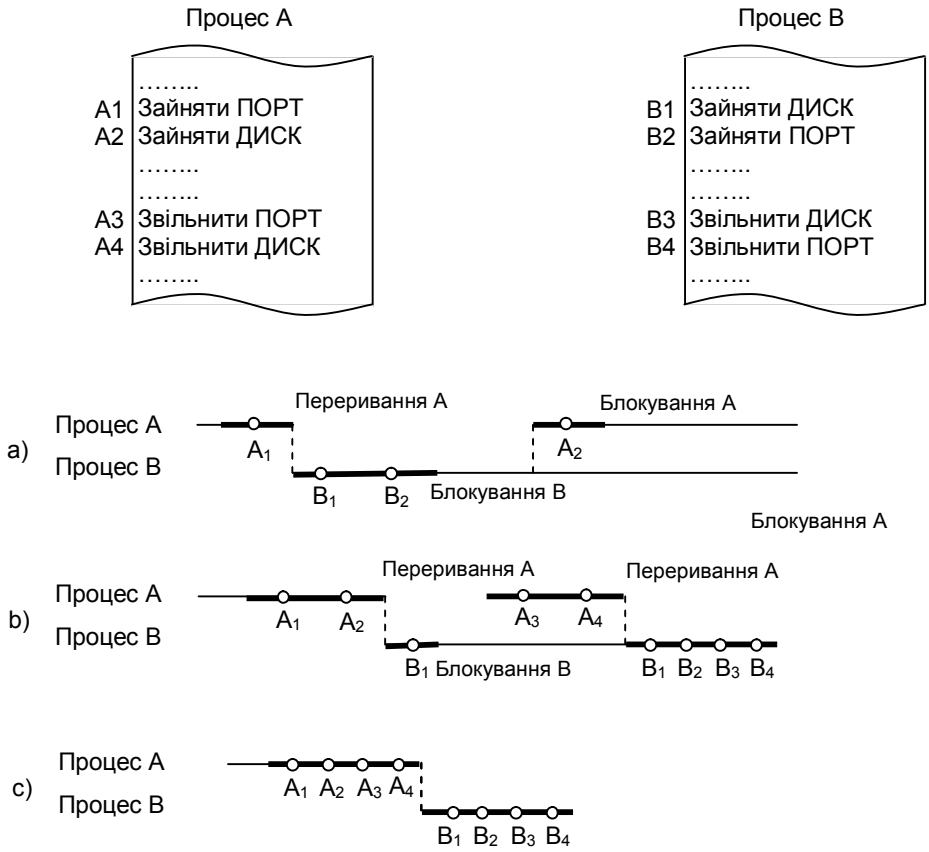


Рис. 3.1. Виникнення взаємного блокування

Залежно від співвідношення швидкостей процесів, вони можуть або взаємно блокувати один одного (дедлок а, б), або утворювати черги до поділюваних ресурсів, або зовсім незалежно використати поділювані ресурси (с).

Програма повинна задовольняти запити таким чином, щоб процеси могли закінчитися, і не виникало тупиків.

Нехай система складається із трьох процесів і десяти пристроїв. Кожному процесу відповідає його максимальна потреба в пристроях, кількість їх, виділена процесу в даний момент, і кількість, що він ще має право запросити.

На рис. 3.2. а) зображено певний стан системи, а на б) – стан системи після зміни. Припустимо, процес С запросив ще два пристрої, і його задовольнили. Тоді, якщо один із процесів запросить максимально можливу для нього кількість пристроїв або більше одного для завершення, буде дедлок.

а)

Ім'я процесу	Максимальна потреба	Виділено	Залишок
A	4	2	2
B	6	3	3
C	8	2	6

б)

Ім'я процесу	Максимальна потреба	Виділено	Залишок
A	4	2	2
B	6	3	3
C	8	4	4

Рис. 3.2. Стани системи

Тому задовольняти запит С небезпечно.

Для визначення того, чи приведе задоволення запиту до небезпечного стану, існують спеціальні алгоритми. Один з них має назву “Алгоритм банкіра”.

Кожному процесу поставлено у відповідність ціле число i (i = 1 ÷ N). Процесу і відповідають його максимальні потреби в пристроях МАКС(i), кількість виділених йому в цей момент пристроїв – ОТРИМ(i), потрібний ще йому залишок пристроїв – ЗАЛИШОК(i). Також, ознака ЗАВЕРШ(i) = 1, якщо процес може завершитися успішно, і дорівнює 0 – якщо ні.

Система заводить глобальні змінні ЗАГПР, що визначають загальне число наявних у системі пристроїв, та ВІЛЬНПР – вільних на даний момент пристроїв.

На початку роботи невідомо, чи може якийсь процес закінчитися

`МОЖЕ НЕ ЗАКІНЧИТИ(i):= true` для всіх i

Щоразу, коли якийсь ЗАЛИШОК може бути виділений із числа незайнятих пристроїв, передбачається, що відповідний процес працює, поки не закінчиться, а потім його пристрої звільняються.

Якщо стан системи стає небезпечним, тоді вона не задовольняє відповідний запит.

```
ВІЛЬНПР:=ЗАГПР;
for i:=1 to Ndo
   Begin
        ВІЛЬНПР:=ОТРИМ(i);
        ЗАЛИШОК(i):=МАКС(i)- ОТРИМ(i);
```

```
         ЗАВЕРШ(i):=false {Процес може не завершитися}
   end;
flag:= true;{Ознака продовження аналізу}
while flag  do
    Begin
      flag:= false;
         for i:=1 to N do
           Begin
            if(not(ЗАВЕРШ(i)and(ЗАЛИШОК(i)<=ВІЛЬНПР)
            then
               Begin
                 ЗАВЕРШ(i):=true;
                 ВІЛЬНПР:=ВІЛЬНПР +ОТРИМ(i);
                 flag:=true
                 end
                 end
         end;
if ВІЛЬНПР = ЗАГПР then
(СТАН СИСТЕМИ БЕЗПЕЧНИЙ, МОЖНА ВИДАТИ ПРИСТРІЙ)
else
(СТАН СИСТЕМИ НЕБЕЗПЕЧНИЙ, ПРИСТРІЙ ВИДІЛЯТИ НЕ МОЖНА)
end.
```

3.2. Проблема розподілу ресурсів та запобігання тупиків

Попередньо при вивченні процесів вирішувались проблеми їх синхронізації та боротьби зі спеціальним станом, який визначили як тупик або дедлок. Це було пов’язано з проблемою сумісного використання деяких ресурсів, що забезпечують виконання обчислювального процесу.

Взагалі-то поняття ресурсів системи узагальнюють та поділяють їх на два класи:

– повторного використання (або системні);

– витратні (або одноразового використання).

У літературі їх умовно позначають як SR (System Resource) та CR (Consumable Resource).

Ці два види ресурсів характеризуються певними особливостями:

1. Системні ресурси SR:

- кількість одиниць ресурсу є константою;
- кожна одиниця ресурсу може бути доступною або виділена одному й тільки одному процесу на деякий час;

- процес може звільнити одиницю ресурсу або зробити її доступною у тому випадку, якщо він її раніше отримав; тобто ніякий процес не може впливати ні на який ресурс, якщо він йому не належить.

Стосовно проблеми тупиків, то у якості SR ресурсів можуть розглядатися: основна пам’ять, зовнішня пам’ять, периферійні пристрої, процесори, файли даних і т. п.

2. Витратні ресурси CR відрізняються від ресурсів SR у декількох важливих відношеннях:

- кількість доступних одиниць ресурсу типу CR змінюється по мірі використання, і звільнюються їх окремі елементи. Кількість одиниць ресурсу є потенційно невичерпною: процес “виробник” збільшує число одиниць ресурсу, вивільняючи одну або більше одиниць ресурсу, які він створив;
- процес “користувач” зменшує число одиниць ресурсу, спочатку вимагаючи, а потім використовуючи одну або більше одиниць. Використані одиниці ресурсу, у загальному випадку, не повертаються ресурсу, а використовуються.

Такі властивості притаманні багатьом сигналам синхронізації, повідомленням та даним, що утворюються апаратурою або програмно.

Для вивчення проблеми тупиків, пов’язаних з вивченням паралельних процесів, розроблено декілька підходів. Одним з них є модель повторно використовуваних ресурсів Холта. Відповідно до цієї моделі, система розглядається як множина процесів та набір ресурсів, причому кожен з ресурсів складається з деякої фіксованої кількості одиниць.

Кожен процес може змінити стан системи за допомою запиту отримання або відмови від ресурсу.

У графічній формі процеси та ресурси позначають квадратиками та кружками відповідно.

Кожний кружок має деяку кількість маркерів, що відповідає числу одиниць цього ресурсу. Дуга, що йде від процесу до ресурсу, означає запит

однієї одиниці цього ресурсу. Дуга, що йде від ресурсу до процесу, означає виділення ресурсу процесу.

Оскільки кожна одиниця SR ресурсу може бути виділена одночасно не більше ніж одному процесу, то кількість дуг, що виходять з процесу до різних ресурсів, не може перевищувати загальної кількості одиниць цього ресурсу.

Така модель зветься графом повторно використовуваних ресурсів.

Наприклад:

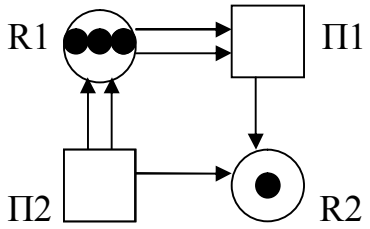


Рис. 3.3. Граф повторно використовуваних ресурсів

Нехай процес П2 запитує дві одиниці ресурсу R1 і одну одиницю ресурсу R2. Процесу П1 належать дві одиниці ресурсу R1, та йому потрібна одна одиниця ресурсу R2.

Якщо процес П2 отримає тепер одиницю ресурсу R2, і прийнято правило, згідно з яким, для звільнення хоча б однієї одиниці якогось ресурсу він має отримати всі потрібні йому ресурси, це призведе до тупикової ситуації: П1 не зможе продовжити роботу, доки П2 не звільнить одиницю ресурсу R2, а П2 не звільнить одиницю R2, доки П1 не поверне дві одиниці R1.

Ясно, що причиною такого дедлоку є неупорядковане використання ресурсів.

Запобігання тупиків зумовлено дуже великою вартістю такого стану, тому він має бути виключений. Запобігання треба розглядати як заборону виникнення цих станів.

Запобігання тупиків має гарантувати, що жодна з нижче перерахованих ситуацій не зможе виникнути:

1. **Ситуація взаємного виключення.** Можна запобігти шляхом дозволу необмеженого поділу ресурсів. Це легко виконати для програм із повторним входом, але зовсім не підходить до сумісно використовуваних змінних у критичних інтервалах.

2. **Ситуація очікування.** Можна запобігти попереднім виділенням ресурсів. При цьому процес може почати виконання тільки після отримання усіх необхідних ресурсів заздалегідь. Тому значна кількість замовлених паралельними процесами ресурсів повинна не перевищувати можливостей системи. Складнощі виникають ще й тоді, коли попереднє виділення ресурсів досить часто є неможливим, оскільки необхідні ресурси стають відомими процесу тільки після початку його виконання.

3. **Ситуація відсутності перерозподілу.** Для цього в ОС повинен бути передбачений механізм відбирання і перерозподілу ресурсів. Перерозподіл деяких ресурсів, наприклад, процесорного часу, достатньо легко зробити, проте перерозподіл, наприклад, периферійних пристроїв досить проблематичний.

4 **Ситуація кругового очікування.** Цю умову можна виключити забороною утворення ланцюгів запитів, що забезпечується шляхом ієрархічного виділення ресурсів. Всі ресурси мають деяку ієрархію. Процес, що запросив ресурси на одному рівні, згодом може запросити ресурси тільки на більш високому рівні. Він може звільнити ресурси на даному рівні тільки після вивільнення усіх ресурсів на всіх більш високих рівнях. Тільки після того, як процес отримав, а потім звільнив ресурси даного рівня, він може запросити ресурси на тому ж самому рівні.

Наприклад, є два види ресурсів: R1 та R2, причому R2 знаходиться на більш високому рівні. Є також 2 процеси П1 і П2. Якщо П1 захопив R1, то П2 не зможе захопити R2, оскільки доступ до нього проходить через R1, який вже отримав П1. Таким чином, замкнене коло виключається.

Обхід тупиків. Його можна інтерпретувати як заборону входження у небезпечний стан. Приклад – алгоритм банкіра. Цей алгоритм має багато недоліків:

1. Вважається, що кількість ресурсів, що підлягають поділу, є сталою величиною.

2. Вимагає попередньої вказівки процесами своїх максимальних потреб у ресурсі. Часто це неможливо.

3. Кількість працюючих процесів має залишатись незмінною. Це нереально, особливо у мультипрограмних системах.

Знаходження тупика за допомогою редукції графа повторно використовуваних ресурсів.

1. Граф повторно використовуваних ресурсів зменшується процесом Пі, який не є ані заблокованим, ані ізольованою вершиною, шляхом видалення усіх дуг, що входять та виходять з Пі. Ця процедура еквівалентна отриманню процесом Пі деяких ресурсів, на які він раніше видавав запити, а потім звільняв усі ресурси. Тоді вершина Пі стає ізольованою.

2. Граф повторно використовуваних ресурсів не редукується (не скорочується), якщо він не може бути скороченим жодним процесом.

3. Граф ресурсів типу SR є повністю скорочуваним, якщо існує послідовність скорочень, яка видаляє усі дуги графу.

Існує лема. Для ресурсів SR порядок скорочень є несуттєвим. Послідовності призводять до одного й того ж нескорочуваного у подальшому графу.

Теорема про тупик. Стан S є станом тупику тоді й тільки тоді, якщо граф повторно використовуваних ресурсів у стані S не є повністю редукованим.

Висновок 1. Процес Пі не знаходиться у тупику тоді й тільки тоді, коли деяка серія скорочень призводить до стану, у якому Пі не заблоковано.

Висновок 2. Якщо S є станом тупику з ресурсами SR, то, у крайньому випадку, два процеси знаходяться у тупику в S.

Із цієї леми безпосередньо випливає алгоритм пошуку тупику. Треба просто спробувати скоротити граф як можна ефективніше. Якщо граф повністю не скорочується, то стан, що був до скорочення, був станом тупику для тих процесів, вершини яких залишились у нескорочуваному графі.

Приклад тупику з ресурсами типу SR розглянуто. Тепер роглянемо приклад тупику з ресурсами CR.

Нехай існують три процеси П1, П2 та П3, які генерують повідомлення М1, М2 та М3. Ці повідомлення не що інше, як CR ресурси. Нехай П1 “користувач”отримує повідомлення М3, процес Р2 отримує повідомлення М1, а П3 – повідомлення М2 від процесу П2, тобто кожен процес є і ”виробником”, і ”користувачем” одночасно, утворюючи кільце. Припустимо, що вони спілкуються через поштові скриньки (ПС), як на рис. 3.4.

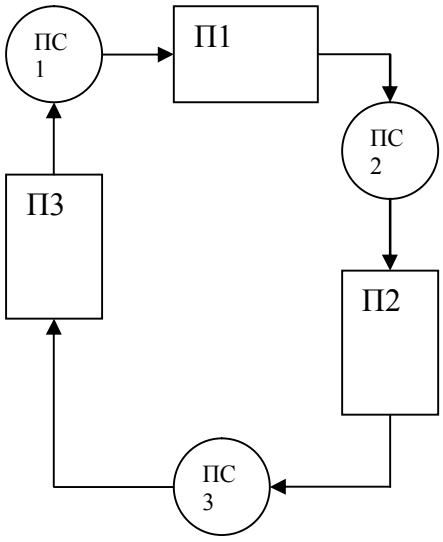


Рис. 3.4. Приклад тупику з ресурсами CR

Якщо цей зв’язок за допомогою повідомлень відтворюється відповідно до наступного порядку:

П1: ……..

Відправити повідомлення (П2, М1, ПС2)

Чекати повідомлення (П3, М3, ПС1)

……..

П2: ……..

Відправити повідомлення (П3, М2, ПС3)

Чекати повідомлення (П1, М1, ПС2)

……..

П3: ……..

Відправити повідомлення (П1, М3, ПС1)

Чекати повідомлення (П2, М2, ПС3)

……..

тоді все гаразд. Проте, переставивши місцями ці дві процедури, прийдемо до тупику:

П1: ……..

Відправити повідомлення (П3, М3, ПС1)

Чекати повідомлення (П2, М1, ПС2)

……..

П2: ……..

Відправити повідомлення (П1, М1, ПС2)

Чекати повідомлення (П3, М2, ПС3)

……..

П3: ……..

Відправити повідомлення (П2, М2, ПС3)

Чекати повідомлення (П1, М3, ПС1)

……..

Дійсно, жоден процес не може відправити повідомлення до тих пір, поки сам його не отримає, а такої ситуації ніколи не може бути.

Приклад тупику на ресурсах CR та SR. Нехай П1 має обмінятися повідомленням з П2, і кожен з них запитує деякий ресурс R, причому П1 потребує три одиниці ресурсу для роботи, а П2 – дві одиниці та тільки на час

обробки повідомлення. Усього ж у розпорядженні є тільки 4 одиниці ресурсу R.

Запит ресурсу можна реалізувати через відповідній монітор з процедурами Request (R, N) – запит N одиниць ресурсу R та Release (R, N) звільнення та повернення N одиниць ресурсу R. Обмін повідомленнями будемо виконувати через поштову скриньку (ПС). Наведемо фрагменти програми:

```
П1: Request (R, 9);
    ……
    Send message (П2, повідомлення, ПС);
    Wait answer (відповідь, ПС);
    ……
    Release (R,3);

П2: Wait message (П1, повідомлення, ПС);
    ……
    Request (R,2);
    Обробка повідомлення;
    ……
    Release (R,2);
    Send answer (відповідь, ПС);
```

Ці два процеси завжди будуть потрапляти в тупик. П2, якщо він буде включений першим, спочатку чекає на повідомлення від П1, після чого буде заблокований при запиті ресурсу R, частина якого вже була віддана П1. П1, отримавши частину ресурсу R, буде також заблоковано в очікуванні відповіді, яку ніколи не отримає, оскільки для цього необхідно отримати ресурс R, який віддано у розпорядження П1.

Тупику можна уникнути лише у тому випадку, коли на час очікування відповіді від П2, П1 буде віддавати хоча б одну одиницю ресурсу R, яким він володіє. У даному випадку, причиною тупику є похибка програмування.

Які ж існують формальні моделі для вивчення тупикових ситуацій? Взагалі моделей дуже багато. І, якщо спеціально вивчати цей предмет, то знадобилося б дуже багато часу. Тому зупинимось на найбільш розповсюдженій моделі – сітці Петрі (окрім вже розглянутої моделі Холта).

3.3 Сітка Петрі

Сітка Петрі була запропонована у 1962 р. Карлом Петрі для моделювання асинхронних потоків інформації у системах перетворення даних.

Взаємодію подій у паралельних асинхронних дискретних системах описують як ситуації, при яких деяка подія може статися. При цьому глобальні ситуації у системі формуються за допомогою локальних операцій, які називають умовами реалізації подій.

Визначений набір умов дозволяє реалізуватися деякій події (передумові), а реалізація події змінює деякі умови (постумовні події). Тобто події взаємодіють з умовами, а умови з подіями.

Формальний механізм сітки Петрі був використаний Холтом.

Існує декілька формальних представлень мереж Петрі:

- теоретично-множинне;
- графова – біхроматичне (орієнтований граф);
- матричне.

Ці сітки можуть бути використані з точки зору аналізу системи на можливість виникнення тупикових ситуацій. Цей аналіз використовується за допомогою дослідження простору можливих станів мережі. Й найчастіше для цього використовують графову модель. Цей підхід базується на побудові редукованого до дерева графа можливих маркувань. У такому дереві

вершини графа – це стани, а гілки, що помічені відповідними переходами, – можливі зміни станів сітки, тобто виконання її переходів.

Якщо взяти довільну вершину такого дерева (за винятком початкової), то шлях до цієї вершини від кореня дерева (від початкового маркування до заданного маркування) буде послідовністю виконання переходів.

Кажуть, що перехід t_j для розмітки М "живий", якщо для усіх розміток $M'(M)$ існує послідовність виконання переходів, яка призводить до маркування M', при якому перехід t_j може спрацювати.

Сітку Петрі називають "живою", якщо всі її переходи "живі". "Живуча" розмітка – це така, при якій кожний з її переходів може запускатися безкінечну кількість разів. Коли досягнута така розмітка, при якій жодний перехід не може бути запущений, кажуть, що сітка Петрі завершилась (досягнута бажане кінцеве маркування) або ж зависла (має місце тупікова ситуація).

У графічному представленні сітки (рис. 3.5), переходи зображено вертикальними або горизонтальними "лініями", а позиції – "кружечками". Умови – позиції та події-переходи, пов'язані відношеннями безпосередньої залежності (безпосереднього причинно-наслідкового зв'язку), як зображено за допомогою направлених дуг, що ведуть з позицій до переходів і переходів до позицій. Позиції, з яких ведуть дуги на даний перехід, звуть вхідними позиціями, а позиції, на які ведуть дуги з даного переходу – вихідними позиціями.

Виконання умови здійснюється розміткою відповідної позиції, власне поміщенням числа N або зображенням N маркерів (фішок) у те місце, де $N > 0$ – ємкість умови.

Переміщення маркерів сіткою здійснюється за допомогою виконання її переходів. Виконання збудженого переходу призводить до зміни маркування сітки, тобто до зміни її стану.

Якщо для сітки M_0 задано початкове маркування, при якому хоча б один перехід збуджено, то у ній починається рух маркерів.

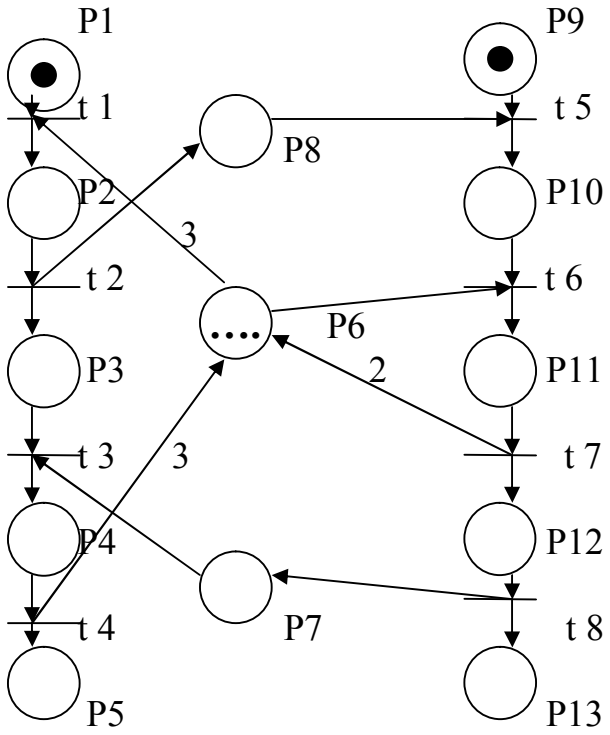


Рис. 3.5. Приклад сітки Петрі для двох взаємодіючих процесів

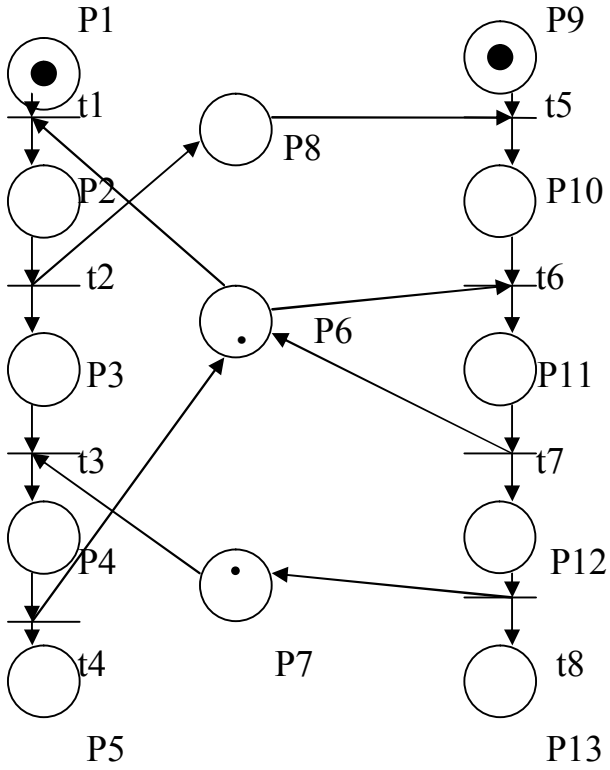


Рис. 3.6. Приклад тупикової ситуації

Число маркерів, які перехід t_j виймає зі своїх вихідних позицій, може не дорівнювати числу маркерів, які цей перехід переміщує у свої вихідні позиції, оскільки зовсім не обов'язково, щоб кількість вхідних дуг переходу дорівнювала кількості його вихідних дуг.

Початкове маркування (1, 0, 0, 0, 0, 4, 0, 0, 1, 0, 0, 0, 0). Тут позиція P2 означає, що P1 отримав три одиниці ресурсу R. Дуга, що поєднує позицію P6 (число в ній відповідає кількості доступних одиниць ресурсу R), має вагу 3, при спрацюванні переходу t1 процес P1 отримає три одиниці ресурсу. Перехід t2 відповідає відправленню повідомлення P2. Перехід t5 – прийом. Поява маркеру у позиції P7 означає, що P2 обробив та послав відповідь P2. Виконання переходу t4 – це повернення 3 одиниць ресурсу, якими володів P1.

Сітка на рис.3.6. не буде "живою", бо в ній будуть "мертві" переходи: t2, t3, t6, t7, t8 (тупикова ситуація).

Розглянемо роботу з сіткою Петрі.

Нехай це сітка Петрі, як на рис. 3.7. Розмітка M сітки – це функція, що ставить у відповідність міткам позицій невід'ємні цілі числа. Розмітка приписує кожній позиції деяку кількість міток відповідно до функції розмітки.

Нехай P – множина позицій в N, а n (P) – число позицій в P. Кожна позиція в N однозначно зв'язана з набором номерів {1, 2, 3, … ,n(P)}.

Розмітку M можна представити як вектор з n(P) елементів, у якому і-ий елемент визначає кількість міток в і-ій позиції.

Розмітка змінюється за умови запуску переходу. Перехід запускається, якщо для кожної позиції x, з якої стрілка вказує на t, існує хоча б одна мітка (якщо стрілка одинична).

На рис. 3.7. переходи a та b можна запустити, M = [2,0,0]. Якщо зробити послідовність запусків a, b, отримаємо: M = [1,1,0]. Тепер можна запустити переходи b, c. Якщо запустити c, отримаємо M = [2,0,0]. Можна, наприклад, запустити a та b одночасно, тоді буде тупик. M = [0,1,1]. Далі

“живих” переходів немає. Якщо запустити b, буде M = [1,0,1]. Далі запускаємо d, отримуємо M = [2,0,0]. Мережа “жива”.

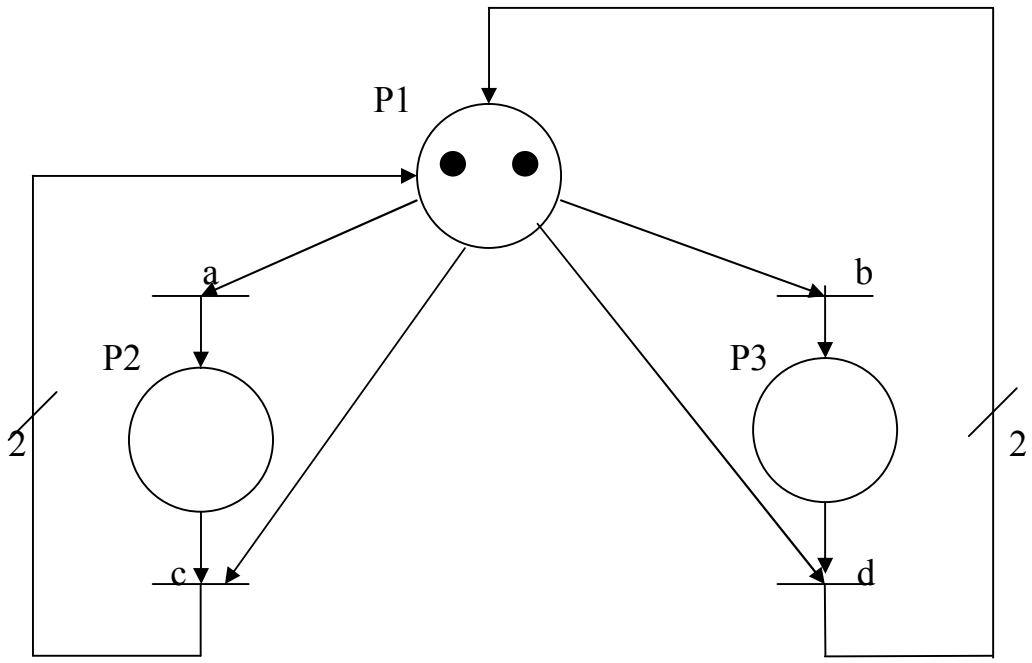


Рис. 3.7. Приклад сітки Петрі

3.4 Контрольні запитання до розділу 3

1. Дайте визначення стану тупику.
2. Вкажіть напрямки політики виявлення та запобігання тупиків.
3. Опишіть алгоритм банкіра.
4. Як пов’язані процеси розподілу ресурсів і запобігання тупиків?
5. На які класи поділяють ресурси?
6. Дайте визначення моделям повторно використовуваних і витратних ресурсів.
7. Які існують методи виявлення і запобігання тупикових ситуацій?
8. Як застосувати сітку Петрі для побудови моделей виявлення та запобігання тупикових ситуацій?

Розділ 4. Мультипрограмування. Розподіл часу процесора

4.1 Основні визначення і характеристики

Час процесора завжди був найважливішим з ресурсів системи, що підлягають розподілу.

Мультипрограмування або багатозадачність – спосіб організації обчислювального процесу, при якому на одному процесорі почергово виконуються одразу декілька програм.

Найбільш характерними критеріями ефективності обчислювальних систем є:

- Пропускна здатність – кількість завдань, виконуваних обчислювальною системою в одиницю часу.
- Зручність роботи користувачів – користувачі мають можливість інтерактивно працювати одночасно з декількома додатками на одній машині.
- Реактивність системи – здатність системи витримувати заздалегідь задані інтервали часу між запуском програми й одержанням результату.

Залежно від цього ОС поділяють на:

- Системи пакетної обробки;
- Системи поділу часу;
- Системи реального часу.

Системи пакетної обробки

Призначалися для рішення завдань обчислювального характеру, що не потребує швидкого одержання результатів. Головною метою й критерієм ефективності систем пакетної обробки є максимальна пропускна здатність.

Для досягнення цієї мети формується **пакет** завдань, кожне завдання містить вимоги до системних ресурсів, із цих завдань формується мультипрограмна суміш, тобто безліч одночасно виконуваних завдань.

Для пакетних ОС характерно поєднання операцій введення-виведення й обчислень. Таке поєднання може досягатися різними способами:

Спеціалізований процесор введення-виведення

Іноді такі процесори називають каналами. Канал має систему команд, що відрізняється від системи команд центрального процесора. Ці команди спеціально орієнтовані на керування зовнішніми пристроями, наприклад :

- установити магнітну головку;
- надрукувати рядок тощо.

Канальні програми можуть зберігатися в тій самій оперативній пам'яті, що й програми центрального процесора. В системі команд центрального процесора передбачається спеціальна інструкція, за допомогою якої каналу передаються параметри й вказівки на програму введення-виведення, яку він повинен виконати (рис. 4.1).

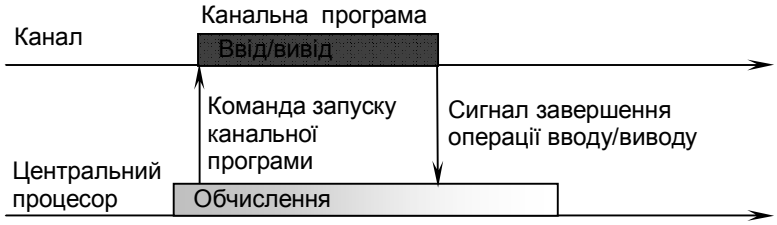


Рис. 4.1. Спеціалізований процесор введення-виведення

Зовнішні пристрої, керовані контролерами. Кожний зовнішній пристрій (або група) має свій власний контролер, який опрацьовує команди від центрального процесора. Контролер і ЦП працюють асинхронно. Контролер повідомляє ЦП про свою готовність прийняти наступну команду сигналом переривання або ЦП довідається про це, періодично опитуючи стан контролера (рис. 4.2).

Максимальний ефект при пакетній обробці досягається при найбільш повному перекритті обчислень і введення-виведення.

У випадку одного завдання, прискорення залежить від його характеру. При перевазі обчислень або введення-виведення прискорення практично відсутнє.

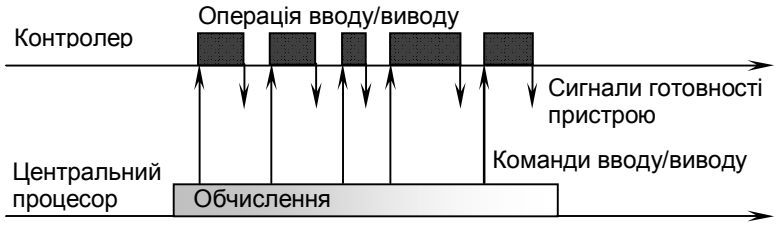


Рис. 4.2. Управління контролером

Системи поділу часу

Системи поділу часу усувають основний недолік пакетної обробки – ізоляцію користувачів-програмістів від процесу виконання їх завдань. Кожному користувачеві виділяють окремий термінал, з якого можна вести свій діалог із програмою. У системах цього типу кожному завданню виділяється квант процесорного часу, жодне завдання не займає процесор надовго. Створюється ілюзія, що процесор належить тільки завданню (або програмістові).

Системи реального часу

Основний критерій – здатність системи витримати заздалегідь задані інтервали часу між запуском програми й одержанням результату. Цей час називають **часом реакції системи,** а відповідну властивість системи – реактивністю. Вимоги вчасної реакції визначаються зовнішніми факторами (наприклад, специфікою системи керування).

У системах реального часу мультипрограмна суміш являє собою фіксований набір заздалегідь розроблених програм, а вибір програми на виконання здійснюється за перериваннями (наприклад, виходячи з поточного стану об'єкта керування) або згідно до розкладу робіт.

Мультипроцесорна обробка

Мультипроцесорна обробка – спосіб організації обчислювального процесу в системах з декількома процесорами, при якому кілька завдань можуть одночасно виконуватися на різних процесорах системи.

На даний час стало звичайним явищем включення декількох процесорів в архітектуру персонального комп'ютера.

Функції підтримки мультипроцесорної обробки даних є у багатьох ОС, в тому числі й таких, як Windows NT.

Мультипроцесорні системи визначають як **симетричні** або як **несиметричні**, в залежності від того, до якого аспекту обчислювальної системи це відноситься:

- до архітектури;
- до способу організації обчислювального процесу.

Симетрична архітектура дозволяє однорідність всіх процесів й однаковість включення процесорів у схему мультипроцесорної системи. Традиційні симетричні мультипроцесорні конфігурації розділяють одну велику пам'ять між всіма процесорами.

Масштабованість або можливість нарощування числа процесорів у симетричних системах обмежена внаслідок того, що всі вони користуються однієї й тією ж оперативною пам'яттю та розташовуються в одному корпусі. Це **масштабування по вертикалі**.

В симетричних архітектурах всі процеси користуються однієї й тією ж схемою відображення пам'яті. Вони можуть швидко обмінюватися даними. Забезпечується висока продуктивність для завдань, які активно між собою взаємодіють (наприклад, при роботі з базами даних).

В **асиметричній архітектурі** процесори можуть відрізнятися як своїми характеристиками, так і функціональною роллю, що надається їм у системі.

Масштабування в асиметричній архітектурі реалізується інакше, ніж у симетричній. Система може складатися з декількох пристроїв, кожний з яких містить один або кілька процесорів. Це **масштабування по горизонталі**.

Кожен такий пристрій називають **кластером**, а всю систему звичайно називають **кластерною**. Спосіб організації обчислювального процесу в мультипроцесорній системі визначається ОС.

Асиметричне мультипроцесування є найпростішим способом організації. Цей спосіб іноді називають «ведучий - ведений».

На «провідному» процесорі працює ОС, що управляє всіма іншими «веденими» процесорами. Він бере на себе функції розподілу завдань і ресурсів, а «ведені» працюють тільки як оброблюючі пристрої, й ніякі дії з організації роботи обчислювальної системи не виконують. Така ОС не набагато складніша за ОС однопроцесорної системи.

Асиметрична організація обчислювального процесу може бути реалізована як для симетричної мультипроцесорної архітектури, так і для несиметричної.

Симетричне мультипроцесування як спосіб організації обчислювального процесу може бути реалізовано тільки в системах із симетричною мультипроцесорною архітектурою.

Симетричне **мультипроцесування** реалізується загально для всіх процесорів ОС. Всі процесори рівноправно беруть участь у керуванні обчислювальним процесом й у виконанні прикладних завдань. Наприклад, сигнал переривання від принтера, що роздруковує дані процесу, виконуваного на деякому процесорі, може бути оброблений зовсім іншим процесором. Різні процесори можуть у якийсь момент одночасно обслуговувати як різні, так й однакові модулі ОС. Для цього модулі ОС повинні мати властивість реєнтерабельності (повторної входимості). ОС повністю децентралізована. Як тільки процесор завершує виконання чергового завдання, він передає керування **планувальникові**, що вибирає із загальної системної черги для всіх процесорів завдання, що буде виконуватися на даному процесорі наступним.

У випадку відмови одного із процесорів, симетричні системи порівняно легко реконструюються, що є перевагою перед асиметричними системами.

4.2 Поняття процесу і потоку

Дотепер розглядали процес як деяку неподільну роботу, виконувану обчислювальною системою.

У ряді ОС визначені два типи роботи. Більша одиниця – процес, що вимагає для своєї реалізації дещо дрібніших робіт, і цю дрібну одиницю називають **потоком**.

При реалізації потоків з'являється можливість організації паралельних обчислень в межах процесу.

Справа в тому, що програми, виконувані в межах одного процесу, можуть мати внутрішній паралелізм, що, у принципі, може прискорити час виконання процесу.

Із цього виходить, що поряд з механізмами керування процесами, в ОС потрібний інший механізм розпаралелювання обчислень, який враховував би тісні зв'язки між окремими гілками обчислень одного й того ж додатку.

Для цього у ряді сучасних ОС використовується механізм **багатопоточної обробки**. Вводиться нова одиниця роботи – **потік виконання**, а поняття «процес» до деякої міри змінює вміст.

Поняттю «потік» відповідає послідовний перехід процесора від однієї команди програми до іншої. ОС розподіляє процесорний час між потоками, а процесу ОС призначає адресний простір і набір ресурсів, які спільно використовуються всіма його потоками.

При керуванні процесами ОС використовує два основних типи інформаційних структур:

- дескриптор процесу;
- контекст процесу.

Дескриптор процесу містить таку інформацію про процес, яка необхідна ядру ОС протягом усього життєвого циклу процесу незалежно від того, чи перебуває він в активному або пасивному стані, чи образ процесу перебуває в

оперативній пам'яті або вивантажено на диск. **Образ** – сукупність кодів команд і даних процесу.

Дескриптори процесів об'єднуються в список, що утворює таблицю процесів. Пам'ять виділяється динамічно в області ядра. На підставі інформації з таблиці процесів, ОС здійснює планування й синхронізацію процесів.

У дескрипторі прямо або посередньо (через покажчики) зберігається інформація про стан процесу, про розташування образу процесу, про ідентифікатор користувача, що створив процес, про родинні процеси, про події, появи яких очікує процес та ін.

Контекст процесу містить інформацію, необхідну для поновлення виконання процесу з перерваного місця: вміст регістрів процесу, коди помилок виконуваних процесором системних викликів, інформацію про всі відкриті даним процесом файли і незавершені операції введення-виведення й інші дані, що характеризують стан обчислювальної системи в момент переривання.

Контекст, так само як і **дескриптор**, доступний тільки програмам ядра, тобто **перебуває у віртуальному адресному просторі ОС**.

Протягом існування процесу виконання його потоків може бути багаторазово перерване й продовжено (далі будемо вважати, що все сказане про потоки, буде відноситися до процесів у цілому, якщо ОС не підтримує потоки).

Перехід від виконання **одного потоку** до іншого здійснюється в результаті **планування й диспетчеризації**.

Роботу з визначення того, у який момент часу необхідно перервати потік і якому саме потоку надати можливість виконуватися, називають **плануванням**.

При плануванні можуть прийматися до уваги **пріоритет потоків**, **час їхнього очікування** в черзі, накопичений час виконання, інтенсивність звертання до пристроїв введення-виведення й ін. фактори.

ОС планує виконання потоків незалежно від того, чи належать вони одному або різним процесам. Так, наприклад, після виконання потоку деякого процесу ОС може вибрати для виконання інший потік того ж процесу або ж призначити до виконання потік іншого процесу.

Планування потоків містить у собі рішення двох завдань:

- визначення моменту часу для зміни поточного активного потоку;
- вибір для виконання потоку із черги готових потоків.

Існує безліч різних алгоритмів планування потоків, які вирішують згадані вище завдання.

Саме особливості реалізації планування потоків найбільшою мірою визначають специфіку ОС, зокрема, чи є вона системою пакетної обробки, системою поділу часу або системою реального часу.

Планування може бути **динамічним** або **статичним**.

При **динамічному плануванні** рішення приймаються під час роботи системи на основі аналізу поточної ситуації.

ОС працює в умовах невизначеності – потоки і процеси з'являються у випадкові моменти часу й також непередбачено завершуються. Динамічні планувальники можуть гнучко пристосовуватися до ситуації, що змінюється. І для пошуку оптимальних рішень ОС повинна витрачати значні зусилля.

Планувальник називають **статичним**, якщо він ухвалює рішення щодо планування не під час роботи системи, а заздалегідь.

Результатом роботи статичного планувальника є таблиця, названа **розкладом**, у якій вказується, якому саме потоку (процесу), коли й на який час повинен надаватися процесор.

Диспетчеризація полягає в реалізації знайденого в результаті планування (динамічного або статичного) рішення, тобто в перемиканні процесора з одного потоку на інший. Перш, ніж перервати виконання потоку, ОС запам'ятовує його контекст, щоб згодом використати цю інформацію для наступного поновлення виконання даного потоку.

Контекст визначає:

- стан апаратури комп'ютера в момент переривання потоку: значення лічильника команд, вміст регістрів загального призначення, режим роботи процесора, прапори, маски й інші параметри;
- параметри операційного середовища (посилання на відкриті файли, дані про незавершені операції введення-виведення, коди помилок, виконувані даним потоком системні виклики тощо).

Диспетчеризація зводиться до наступного :

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

У мультипрограмній системі потік може перебувати в одному із трьох станів:

- виконання – виконується процесором:
- очікування – чекає здійснення деякої події;
- готовність – має всі необхідні для виконання ресурси, готовий до виконання, але процесор зайнятий виконанням іншого потоку.

Зауважимо, що стани виконання й очікування можуть бути віднесені й до завдань, що виконується в однопрограмному режимі, а стан **готовності** характерний тільки для режиму мультипрограмування. Граф станів потоку в багатозадачному середовищі можна представити як на рис. 4.3.

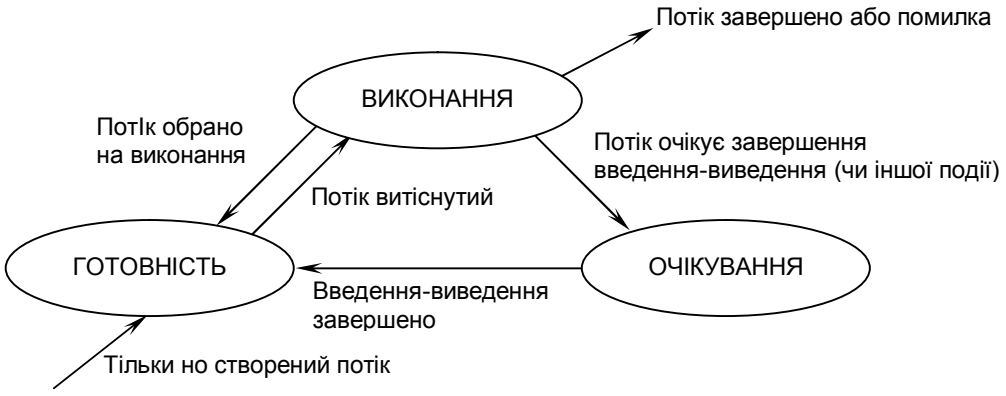


Рис. 4.3. Граф станів потоку в багатозадачному середовищі

Витіснення потоку означає припинення його виконання процесором, наприклад, внаслідок вичерпання відведеного для виконання кванта часу.

У стані виконання в однопроцесорній системі може перебувати не більше одного потоку, а в кожному зі станів очікування й готовності – кілька потоків. Ці потоки утворюють черги потоків, що **очікують,** і **готових** потоків.

4.3 Алгоритми планування

Витісняючі і невитісняючі алгоритми планування

Всю безліч алгоритмів планування можна розділити на два класи: витісняючі і невитісняючі.

Невитісняючі засновані на тому, що активному потоку дозволено виконуватися доти, поки він сам не вирішить віддати керування ОС.

Витісняючі – такі, у яких рішення про перемикання процесора з виконання одного потоку на іншій приймає ОС.

При мультипрограмуванні, що не витісняє, механізм планування розподілено між ОС і прикладними програмами. Прикладна програма, одержавши керування від ОС, сама визначає момент завершення чергового циклу свого виконання й тільки потім передає керування ОС за допомогою якого-небудь системного виклику.

Тому розроблювачі програм (програмісти) для ОС із **невитісняючою** багатозадачністю змушені брати на себе частину функцій планувальника й створювати програми так, щоб вони виконували свої завдання невеликими частинами. Це може бути як недоліком, так і перевагою, якщо, наприклад, наперед відомий набір постійно розв'язуваних завдань.

Майже всі широко відомі ОС, такі як UNIX, Windows NT/2000, OS-2, Windows 95/98 реалізують витісняючі алгоритми планування потоків.

Алгоритми планування, засновані на квантуванні

В основі багатьох витісняючих алгоритмів планування лежить концепція **квантування**. Відповідно до неї, кожному потоку для роботи почергово виділяється обмежений неперервний проміжок часу – **квант**.

Зміна активного потоку відбувається, якщо:

- потік завершився й залишив систему;
- відбулася помилка;
- потік перейшов у стан очікування;
- вичерпано квант процесорного часу.

Потік, що вичерпав свій квант, переводиться в стан готовності й очікує в черзі. Граф станів потоку представлений на рис. 4.4.

Кванти для потоків можуть бути однаковими або різними.

Рис. 4.4. Граф станів потоку

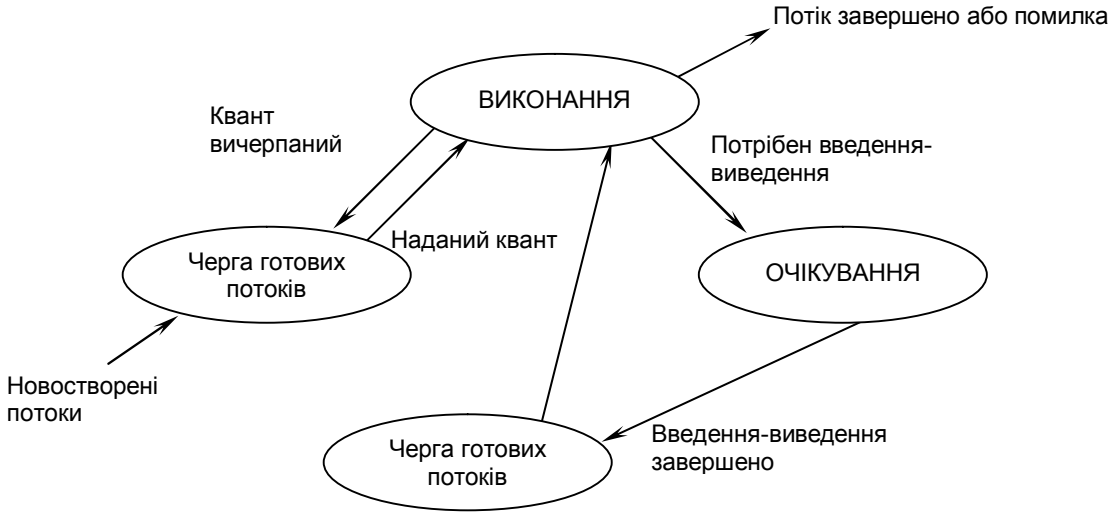


Рис. 4.5. Граф станів потоку з чергами

Черга може бути простою або з пріоритетами. Наприклад, якщо потоки не повністю використовують кванти часу через операції введення-виведення, з них можна утворити пріоритетну чергу як відповідну компенсацію за неповністю використані кванти. Відповідний граф станів потоку можна представити як на рис. 4.5.

Алгоритми планування, засновані на пріоритетах

Пріоритет – це число, що характеризує ступінь привілейованості потоку при використанні ресурсів. У більшості ОС пріоритетів потоку пов'язаний із пріоритетом процесу, в межах якого виконується даний потік. Значення пріоритетів включають в описувач процесу. ОС може змінювати пріоритети потоку залежно від ситуації. В останньому випадку, пріоритети називають **динамічними**, на відміну від незмінних, які називають **статичними** (або фіксованими).

В ОС Windows NT визначено 32 рівні пріоритетів і два класи потоків – потоки реального часу й потоки зі змінними пріоритетами.

Діапазон від 1 до 15 відведено для потоків зі змінними пріоритетами, а від 16 до 32 – для більш критичних до часу потоків реального часу.

Змішані алгоритми планування

У ряді ОС алгоритми планування побудовані з використанням як концепції квантування, так і пріоритетів. Наприклад, в основі планування лежить квантування, але величина кванта й порядок вибору потоків із черги готових визначається пріоритетами потоків.

Так зроблено в Windows NT, у ній квантування поєднується з динамічними абсолютними пріоритетами. На виконання вибирається потік з найвищим пріоритетом. Йому виділяється квант часу. Якщо під час виконання в черзі готових потоків з'являється потік з більш високим пріоритетом, то він витісняє виконуваний потік. Витіснутий потік повертається в чергу готових, причому він стає попереду всіх інших потоків, що мають такий самий пріоритет.

Планування в системах реального часу

У системах реального часу головним критерієм є забезпечення часових характеристик обчислювального процесу.

Будь-яка система реального часу повинна реагувати на сигнали керованого об'єкта протягом заданих часових обмежень. Необхідність ретельного планування робіт полегшується тим, що в системах реального часу весь набір виконуваних завдань відомий заздалегідь.

Крім того, у системі є інформація про час виконання завдань, моментах активації, граничних припустимих строках очікування відповіді й т.д. Ці дані можуть бути використані планувальником для створення статичного розкладу або для побудови адекватного алгоритму динамічного планування.

Якщо наслідки невиконання часових обмежень системою катастрофічні (наприклад, прокатний стан), система називається **жорсткою**. Якщо невиконання обмежень не настільки серйозно (наприклад, система продажу авіаквитків), система називається **м'якою**.

Моменти перепланування

Для реалізації алгоритму планування ОС повинна одержувати керування щоразу, коли в системі відбувається подія, що вимагає перерозподілу процесорного часу. До таких подій відносять наступні:

- переривання від таймера, що сигналізує про закінчення часу, відведеного активному завданню;
- активне завдання виконало системний виклик, пов'язаний із запитом на введення-виведення або на доступ до зайнятого у даний момент ресурсу (наприклад, файлу даних);
- активне завдання виконало системний виклик, пов'язаний зі звільненням ресурсу. Планувальник перевіряє, чи не очікує цей ресурс якогось завдання. Якщо так, то завдання переводиться зі стану очікування в стан готовності й перевіряється, чи має воно найвищий пріоритет. Якщо ні – можливе перепланування;

- зовнішнє апаратне переривання. Воно сигналізує про перехід відповідного поточного завдання до черги готовності й виконується перепланування;

- внутрішнє переривання повідомляє про помилку в поточному завданні. Планувальник знімає завдання й виконує перепланування.

При виникненні кожної з цих подій планувальник виконує перегляд черг і вирішує питання про те, яке завдання буде виконуватися наступним.

Нагадаємо, що стратегія планування визначає, які процеси плануються до виконання, відповідно до поставленої мети.

Відомо досить багато стратегій визначення процесу, якому треба віддати процесор. Найбільш поширеними є наступні:

- якщо можливо, завершити процеси у тому порядку, в якому вони були розпочаті;
- віддати перевагу найбільш коротким за часом виконання процесам;
- надати усім користувачам (тобто процесам) однакові послуги щодо часу очікування у відповідних чергах.

Перелік та класифікацію дисциплін представлено на рис. 4.6.

Усі дисципліни диспетчеризації, незважаючи на їх велику різноматність, можна поділити на два великих класи: безпріоритетні та пріоритетні.

При безпріоритетному обслуговуванні вибір процесу на виконання визначається у деякому наперед встановленому порядку (скажімо, визначеним планувальником) без урахування їх значимості та часових обмежень. Про пріоритети треба запам’ятати наступне:

- пріоритет процесу може бути постійним;
- пріоритет процесу може бути змінено при його виконанні.

При реалізації пріоритетних дисциплін окремим процесам надається перевага право на виконання. Слід завжди пам’ятати, що використання динамічних пріоритетів потребує додаткових витрат на обчислення пріоритетів у певний час, тому у більшості випадків реального часу використовуються методи диспетчеризації на основі фіксованих пріоритетів.

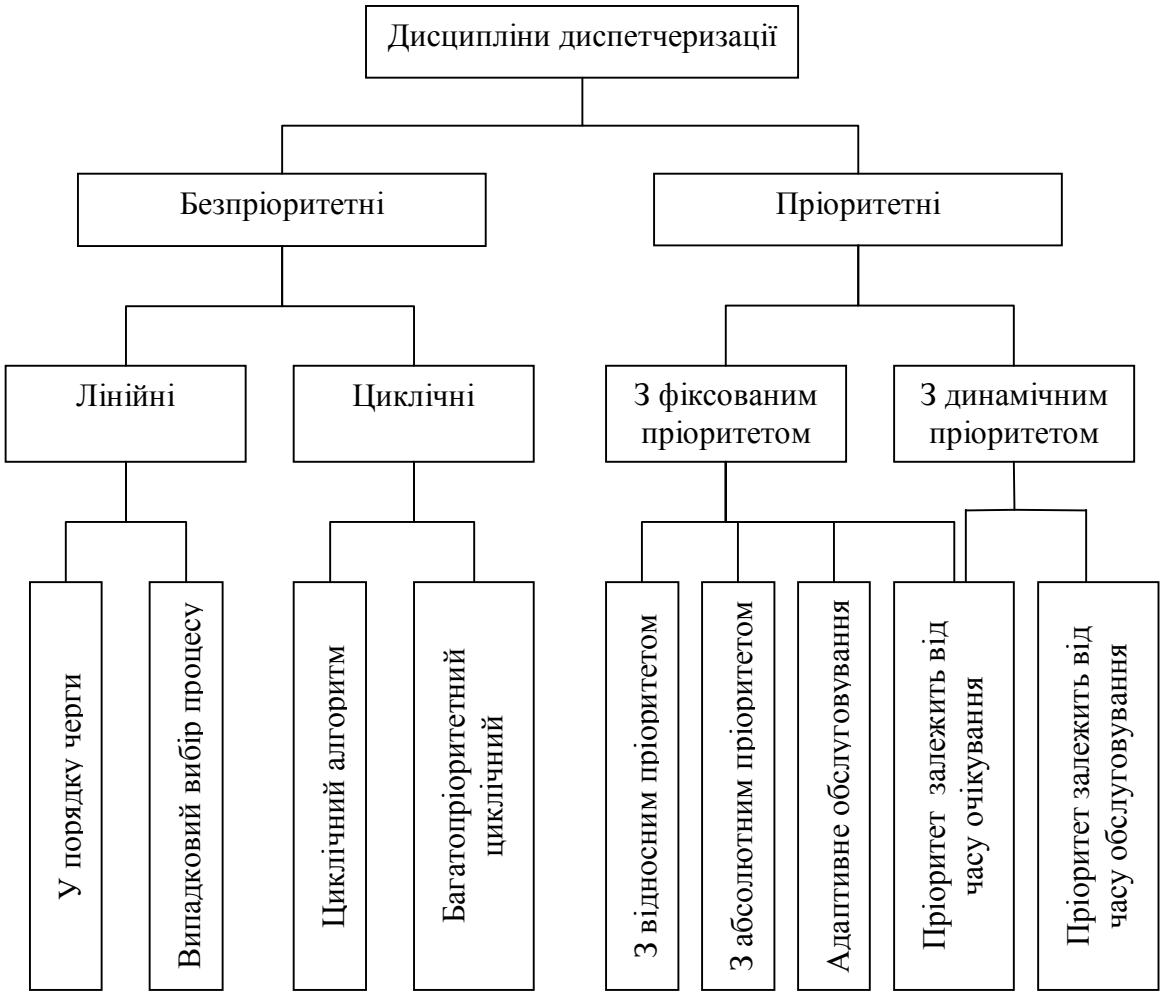


Рис. 4.6. Класифікація дисциплін диспетчеризації

Розглянемо деякі найбільш типові дисципліни диспетчеризації.

Найпростішою є дисципліна FCFS (First Come – First Served), тобто обслуговування у порядку черги. Ті процеси, що були заблоковані в процесі виконання (скажімо, чекають завершення введення-виведення) стають у чергу готовності перед тими процесами, які ще не починали виконуватись. Тобто створюються дві черги: одна з нових процесів, друга – з процесів, що вже виконувались, але перейшли в режим очікування. Такий перехід дозволяє реалізувати тезу про завершення процесів у порядку їх виникнення.

Ця дисципліна обслуговування (рис. 4.7) не потребує втручання в хід виконання процесів, і перерозподіл процесорного часу не відбувається.

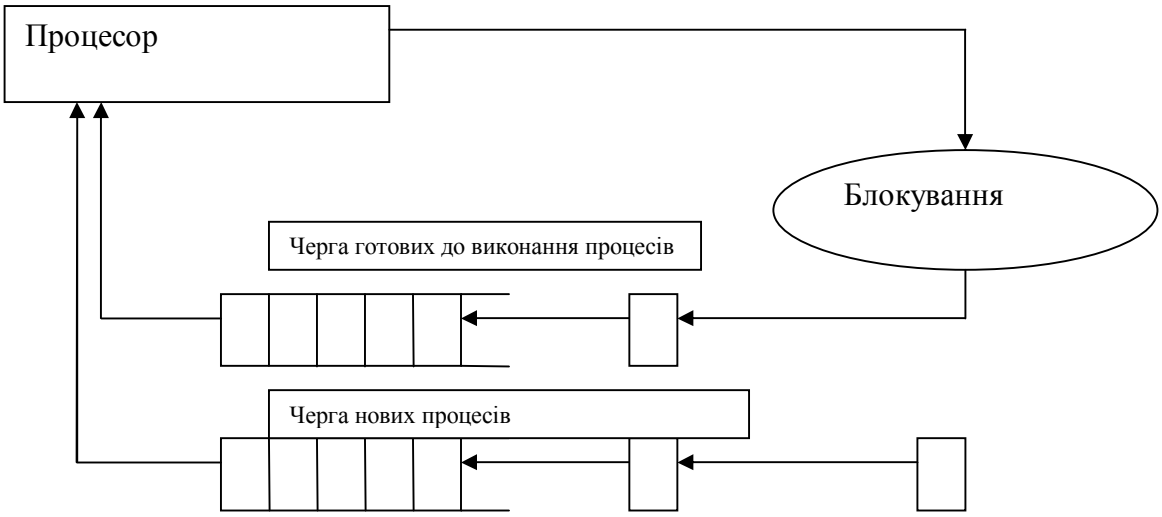


Рис. 4.7. Графічне представлення дисципліни FCFS

Алгоритм FCFS

Детальніше про згаданий алгоритм планування задач, який дістав назву FCFS – першим прийшов, першим обслуговується, можна сказати наступне. Уявимо, що процеси, які знаходяться у стані готовності, створили чергу. Коли черговий процес переходить у стан готовності, він стає в кінець цієї черги. Вибір нового процесу на виконання здійснюється з початку черги.

Цей алгоритм реалізує невитісняюче планування. Процес, що отримав у своє розпорядження процесор, використовує його до закінчення часу свого повного виконання (CPU burst). Після цього обирається новий процес з початку черги.

Перевагою FCFS є простота його реалізації, але у нього є і недоліки. Наприклад, нехай у стані готовності знаходяться три процеси P0, P1 та P3, для яких відомий черговий час їхніх CPU burst. Ці часові характеристики наведені у табл.4.1 у деяких умовних одиницях.

Нехай вся робота процесів обмежується використанням тільки одного CPU birst, вони не використовують операцій вводу – виводу, і час переключення контекстів дорівнює нулю.

Таблиця 4.1. Тривалість процесів

Процес	P0	P1	P2
Тривалість чергового CPU burst	13	4	1

Якщо процеси розташовані у черзі готових до виконання процесів у порядку P0 P1, P2, то їх виконання виглядає як на рис. 4.8.

Рис. 4.8. Виконання процесів P0, P1, P2

Першим на виконання обирається процес P0, який отримує процесор на весь свій час CPU burst, тобто 13 одиниць часу. Після його закінчення виконується процес P1, що займає 4 одиниці часу, потім процес P1 отримує процесор на 1 одиницю часу. Час очікування для процесу P0 складає 0 одиниць часу, для P1 – 13 одиниць, для P2 – 17 одиниць. Середній час очікування у черзі для процесів складає 10 одиниць часу. Повний час виконання для процесу P0 складає 13 одиниць, для P1 – 17 одиниць, для P2 – 18 одиниць. Середній повний час виконання складає 16 одиниць.

Якщо ті ж процеси розміщуються у черзі як P2, P1, P0 (рис. 4.9), то час

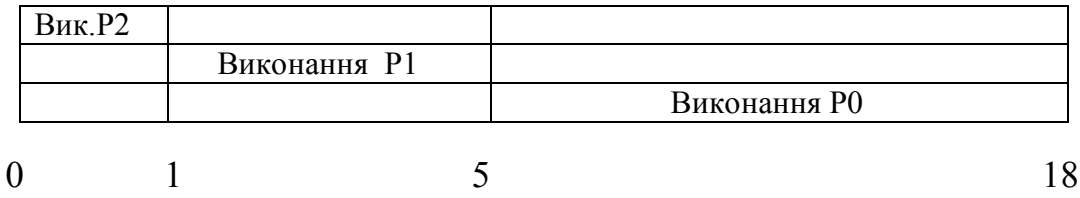


Рис. 4.9. Виконання процесів P2, P1, P0

очікування для процесу P0 становить 5 одиниць часу, для процесу P1 – 1 одиницю, для P2 – 0 одиниць.

Середній час очікування дорівнює 2 одиниці, що у 5 разів менше, ніж у попередньому випадку. Повний час виконання для процесу P0 становить 18 одиниць, для P1 – 6 одиниць, для P2 – 1 одиницю. Середній повний час виконання становить 8 одиниць, тобто в 2 рази менше ніж у попередньому випадку.

Слід зазначити, що середній час очікування і середній повний час виконання процесів для цього алгоритму планування суттєво залежить від порядку розташування процесів у черзі, коли їх CPU burst суттєво різняться.

Перевагами цієї дисципліни є простота реалізації та мали витрати системних ресурсів на формування черги процесів.

Однак, недолік у тому, що при збільшенні завантаження обчислювальної системи зростає середній час очікування обслуговування, причому невеликі за часом виконання процеси мають очікувати стільки ж, як і великі завдання.

Алгоритм RR

Дисципліна обслуговування **RR (Round Robin)** передбачає, що кожний процес отримує процесорний час частками (квантами). Після закінчення кванту процес залишає процесор. Процесор передається іншому процесу. Процес, що був знятий з виконання, стає у чергу готових до виконання процесів. Ця дисципліна проілюстрована рис. 4.10.

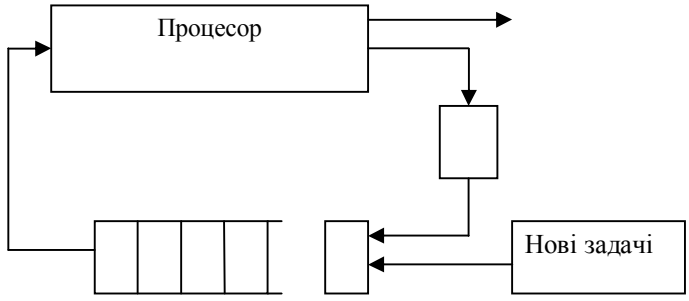


Рис. 4.10. Алгоритм RR

RR ще називають карусельною дисципліною диспетчеризації. Для оптимальної роботи системи необхідно вибрати закон, за яким кванти часу надаються процесам.

Алгоритм **Round Robin** можна розглядати як модифікацію алгоритму **FCFS**. По суті, це той самий алгоритм, але з витісняючою стратегією. Процеси можна уявити собі організованими циклічно, як ніби вони знаходяться на каруселі. Ця карусель крутиться, і кожен з процесів займає процесор певний, наперед заданий квант часу. Величини цього кванту , як правило, обираються у межах 10 – 100 мсек.. Ці процеси, як і у випадку FCFS, знаходяться у черзі готових до виконання процесів. Планувальник обирає для чергового виконання процес у порядку черги і надає йому процесор на час кванту. При виконанні процесу можливі два варіанти:

- час неперервного використання процесору для завершення обраним процесом свого CPU burst менше або дорівнює довжині кванта. Тоді процес звільняє процесор, на виконання обирається новий процес, слідуючий у черзі, і відлік кванту починається з початку;
- довжина залишку CPU burst процесу більша довжини кванта. Тоді при завершенні кванта процес переривається таймером, знімається з виконання і повертається у кінець черги. На виконання обирається наступний, перший за чергою процес, і процедура повторюється.

Повернемось до попереднього прикладу з порядком виконання процесів P0, P1, P2 та з величиною кванту, що дорівнює 4. Виконання процесів може бути проілюстроване табл. 4.2.

Таблиця 4.2. Алгоритм RR (квант = 4)

Час	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P0	В	В	В	В	Г	Г	Г	Г	Г	В	В	В	В	В	В	В	В	В
P1	Г	Г	Г	Г	В	В	В	В										
P2	Г	Г	Г	Г	Г	Г	Г	Г	В									

Позначка “В” говорить про те, що процес знаходиться на процесорі і виконується, а “Г” – стоїть у черзі.

Першим на виконання обирається процес P0. Величина його CPU burst більша ніж величина кванту, і тому він виконується лише частково – 4 одиниці часу (за час кванту), після чого стає назад у чергу готових до виконання процесів. Слідуючим обирається на виконання процес P1. Час його виконання співпадає з довжиною кванту і тому процес P1 повністю виконується. Тепер черга складається лише з процесів P2, P0. Процесор надається процесу P2, і він завершується за одну одиницю часу, після чого завершується процес P0.

Час очікування для процесу P0 складає 5 одиниць часу, для процесу P1 – 4 одиниці, для процесу P2 – 8 одиниць. Середній час очікування, у цьому випадку, становить 5,66 одиниць часу. Повний час виконання для процесу P0 складатиме 18 одиниць часу, для P1 – 8, а для P2 – 9 одиниць. Середній повний час виконання становитиме 11,66 одиниць часу.

Легко впевнитись у тому, що середній час очікування та середній повний час виконання для зворотного порядку розташування у черзі готових до виконання процесів не відрізняється від відповідних величин часу для алгоритму FCFS і складає відповідно 2 і 6 одиниць.

На продуктивність алгоритму RR значно впливає величина кванту часу. Розглянемо попередній приклад, але оберемо величину кванту, що дорівнює 1 (табл. 4.3).

Таблиця 4.3. Алгоритм RR (квант = 1)

Час	1	2	3	4	5	6	7	8	9	11	12	13	14	15	16	17	18
P0	В	Г	Г	В	Г	В	Г	В	В	В	В	В	В	В	В	В	В
P1	Г	В	Г	Г	В	Г	В	Г	В								
P2	Г	Г	В														

При дуже великих розмірах кванту часу, коли кожен процес встигає завершити свій CPU burst до переривання, алгоритм RR вироджується у FCFS. При дуже маленьких значеннях величини кванту часу виникає ілюзія того, що кожен із процесів виконується на окремому віртуальному процесорі,

продуктивність яких нижче у стільки разів, скільки процесів одночасно виконується.

Величина кванту визначається як компроміс між бажаним часом реакції системи на запит користувачів та накладними витратами на часту заміну контексту процесів.

Алгоритм SJF

Попередньо показано, наскільки суттєвим для алгоритмів FCFS і RR є розподілення процесів у чергах готовності за часом CPU birst. Якщо кожного разу черговість виконання процесів перекомпоновувати у порядку зростання їхніх CPU burst, то можна покращити середні характеристики розглянутих процесів.

Отже, якщо у стані готовності процесів два або більше, то на процесор слід обирати процес з найменшим значенням CPU burst. Відповідний алгоритм дістав назву **Shortest Job First** (**SJF**). Цей алгоритм може бути як витісняючим так і невитісняючим.

При **невитісняючому SJF** плануванні процесор надається обраному процесу на увесь потрібний час виконання незалежно від подій, що мають місце у обчислювальній систем. При витісняючому алгоритмі враховується поява нових процесів у черзі готових до виконання. Якщо CPU birst нового процесу менше ніж залишок процесу, що виконується, останній витісняється і починає виконуватись цей новий процес.

Розглянемо роботу невитісняючого SJF. Нехай у стані готовності знаходяться процеси P0, P1,P2, P3 з CPU burst, що наведені у табл. 4.4.

Таблиця 4.4. CPU burst процесів

Процес	P0	P1	P2	P3
Величина чергового CPU burst	5	3	7	1

Першим на виконання буде обрано процес P3. Далі процеси P1, P0 і P2 (див. табл. 4.5).

Таблиця 4.5. Алгоритм SJF

Час	1	2	3	4	5	6	7	8	9	10	12	13	14	15	16
P0	Г	Г	Г	Г	В	В	В	В	В						
P1	Г	В	В	В											
P2	Г	Г	Г	Г	Г	Г	Г	Г	Г	В	В	В	В	В	В
P3	В														

Середній час очікування для алгоритму SJF складає 3,5 одиниці часу. Легко побачити, що для алгоритму FCFS і для процесів у порядку P0, P1, P2, P3, ця величина складатиме 7 одиниць часу, тобто буде у два рази більшою ніж для алгоритму SJF.

Розглянемо витісняючий варіант алгоритму SJF. Для цього візьмемо процеси P0, P1, P2, P3 з різними значеннями CPU burst та різними моментами їх появи у черзі готових до виконання процесів (див. табл. 4.6).

Таблиця 4.6. CPU burst процесів

Процес	Час появи у черзі	Тривалість чергового CPU burst
P0	0	6
P1	2	2
P2	6	7
P3	0	5

У нульовий момент часу у стані готовності знаходяться тільки два процеси – P0 і P3. Менший час чергового CPU burst у процесу P3. Він і обирається на виконання (див. табл. 4.7).

Через дві одиниці часу надходить процес P1. Час його CPU burst менший ніж залишок у процесу P3. Він знімається з виконання і переводиться у чергу готових до виконання процесів. Через дві одиниці часу процес P1 завершується, і для виконання знову обирається процес P3.

Таблиця 4.7. Алгоритм SJF з витісненням

Час	1	2	3	4	5	6	7	8	9	10
P0	Г	Г	Г	Г	Г	Г	Г	В	В	В
P1			В	В						
P2							Г	Г	Г	Г
P3	В	В	Г	Г	В	В	В			

Час	11	12	13	14	15	16	17	18	19	20
P0	В	В	В							
P1										
P2	Г	Г	Г	В	В	В	В	В	В	В
P3										

У момент часу 6, у черзі готових процесів з’являється процес P2, але, оскільки йому потрібно для виконання 7 одиниць часу, а процесу P3 до завершення залишилось тільки 2 одиниці часу, він і завершується. Після його завершення у черзі залишаються два процеси – P0 і P3, з яких обирається P0. Останнім завершується процес P2.

Основною складністю використання алгоритму SJF є проблеми з визначенням величин CPU burst процесів під час роботи обчислювальної системи. В системах з пакетною обробкою розміри процесорного часу, що потребує завдання, вказує користувач при формуванні завдання. Якщо користувач заявить більше часу ніж потрібно, він буде чекати результату довше, оскільки завдання буде завантажене на виконання пізніше. Якщо ж він заявить менше часу, заявка може не бути виконаною до кінця. Виключення можуть становити системи реального часу, де наперед відомий час виконання кожної із задач і їх перелік у процесі роботи не змінюється.

Цей недолік нівелюють дисципліни **SJN** та **SRT**.

Дисципліна **SJN (Shortest Job Next)** означає: слідуючим буде виконане найкоротше завдання. Ясно, що при цьому диспетчер має порівнювати заявлений на виконання час та час фактичного виконання, та, у випадку

перевищення фактичного часу, процес має бути поставлено у кінець черги (або може використовуватись інша система штрафів).

Дисципліна SJN визначає, що є тільки одна черга завдань, готових до виконання. Завдання, що були тимчасово заблоковані, знову стають у кінець черги поряд з новими процесами. Це теж призводить до того, що процеси, яким потрібно мало часу для виконання, в кінці-кінців також вимушені очікувати процесор поряд з довшими процесами.

Для усунення цього недоліку була запропонована дисципліна **SRT** (**Shortest Remaining Time**) – слідуюче завдання потребує для завершення найменше часу.

Гарантоване планування

При інтерактивній роботі *N* користувачів слід використовувати алгоритми планування, які б гарантували, що кожен з них отримає у середньому 1/N процесорного часу. Попередьо бачили, що такі гарантії можна, у першому наближенні, забезпечити, використовуючи алгоритм планування **RR.** Тому його дуже часто використовують у подібних цілях.

Пріоритетне планування

Алгоритм SJF являє собою, власне, окремий випадок пріоритетного планування. При пріоритетному плануванні кожному процесу надається певне число – пріоритет, відповідно до якого йому надається процесор.

Процеси з однаковим пріоритетом плануються відповідно до алгоритму FCFS. Для алгоритму SJF у якості такого пріоритету виступає оцінка чергового CPU burst. Чим вона менше, тим вищий пріоритет.

Принципи визначення пріоритетів можуть опиратися як на внутрішні особливості обчислювальної системи, так і на певні особливості обчислювального процесу і його процесів. Планування з пріоритетами може бути як витісняючим, так і невитісняючим. При витісняючому плануванні процес з більш високим пріоритетом витісняє з процесора процес, що

виконується, повертаючи його у чергу готових до виконання процесів. При невитісняючому плануванні він просто стає першим у черзі готових до виконання процесів.

Нехай у системі знаходяться ті ж самі процеси P0, P1, P2, P3 з параметрами як у табл. 4.8.

Таблиця 4.8. Параметри процесів

Процес	Час появи у черзі	Тривалість чергового CPU burst	Пріоритет
P0	0	6	4
P1	2	2	3
P2	6	7	2
P3	0	5	1

Як будуть вести себе процеси при невитісняючому алгоритмі пріоритетного планування, показано у табл. 4.9. Першим на виконання з черги готових для виконання процесів обирається процес P3. Після його завершення у черзі готових до виконання процесів будуть процеси P0 та P1. Більший пріоритет у процесу P1, тому його і обирають на виконання. У момент часу 8 на виконання буде обрано процес P2, а за ним – P0.

Таблиця 4.9. Пріоритетне планування

Час	1	2	3	4	5	6	7	8	9	10
P0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г
P1			Г	Г	Г	В	В			
P2							Г	В	В	В
P3	В	В	В	В	В					
Час	11	12	13	14	15	16	17	18	19	20
P0	Г	Г	Г	Г	В	В	В	В	В	В
P1										
P2	В	В	В	В						
P3										

Надання процесора процесам у випадку витісняючого алгоритму пріоритетного планування буде відрізнятись від попереднього (див. табл. 4.10).

Таблиця 4.10. Пріоритетне планування з витісненням

Час	1	2	3	4	5	6	7	8	9	10
P0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г
P1			Г	Г	Г	В	Г	Г	Г	Г
P2							В	В	В	В
P3	В	В	В	И	В					
Час	11	12	13	14	15	16	17	18	19	20
P0	Г	Г	Г	Г	В	В	В	В	В	В
P1	Г	Г	Г	В						
P2	В	В	В							
P3										

У прикладі, що розглянутий вище, пріоритети процесів не змінювались у часі. Такі пріоритети називають статичними. Механізми статичної пріоритетності легко реалізувати, і це пов’язано з відносно невеликими витратами на вибір найбільш пріоритетного процесу. Однак, статичні пріоритети не реагують на зміни ситуацій у обчислювальній системі, які можуть призвести до того, що зміна пріоритетів була б бажаною. Тому більш гнучкими є динамічні пріоритети, які змінюються під час виконання обчислень системою. Алгоритми планування з динамічною пріоритизацією набагато складніші у реалізації, що пов’язано з великими втратами продуктивності у порівнянні із статичними схемами.

Багаторівневі черги

Для систем, де процеси можуть бути розсортовані на певні групи, розроблено дещо інший клас алгоритмів планування. Для кожної групи процесів створюється своя черга процесів, що знаходяться у стані готовності. Цим чергам визначаються фіксовані пріоритети. У середині цих черг можуть

бути використані які завгодно алгоритми планування. Наприклад, FCFS або RR.

Такий метод дістав назву **багаторівневих черг (Multilevel Queue)**.і дозволяє підвищити ефективність планування, оскільки для різного роду процесів можуть бути одночасно використані різні, найбільш ефективні алгоритми планування. Для більш ефективної роботи до нього додають ще механізм зворотного зв'язку. У цьому випадку, процес постійно не прив'язують до конкретної черги, а дозволяють йому мігрувати з однієї черги до іншої за певних попередньо визначених умов

Наприклад, нехай передбачено дві пріоритетні черги 1 і 2. Черга з номером 1 має вищий пріоритет з алгоритмом планування RR і довжиною кванта 10 мсек. Черга з номером 2 – з меншим пріоритетом і довжиною кванта 20 мсек. Процес, що знаходиться у стані готовності, потрапляє спочатку у чергу 1 і, якщо для виконання йому замало 10 мсек, він залишається у цій черзі, якщо для завершення йому буде достатньо чергового кванту у 10мсек, якщо ні – його переводять у чергу номер 2 і т. д.

Багаторівневі черги із зворотнім зв'язком являють собою найбільш загальний підхід до планування. Але реалізація цього підходу найбільш трудомістка, хоча і є найбільш гнучкою і ефективною.

Зауважимо, що для повного опису такого типу алгоритмів планування слід попередньо визначити:

- кількість черг процесів, що знаходяться у стані готовності;
- алгоритм планування взаємодії між чергами;
- алгоритм планування в середині кожної черги;
- правила розміщення нового готового до виконання процесу в одну із черг;
- правила переходу з однієї черги в іншу.

Для порівняння різних алгоритмів диспетчеризації використовуються наступні критерії:

- заватаженість процесора. Вона змінюється від 2 до 3 відсотків у персональному комп'ютері, і до 90 – 100 відсотків для сильно завантажених серверів;
- пропускна здатність. Вона вимірюється числом процесів, які можуть виконуватись в одиницю часу;
- час обернення (turnaround time). Для деяких процесів важливим критерієм є повний час виконання, тобто інтервал від моменту появи процесу у черзі до моменту його завершення;
- час очікування – сумарний час знаходження процесу у черзі готових процесів;
- час відгуку (response time) – час між постановкою процесу у чергу та моментом першого звернення до терміналу. Цей критерій важливий для інтерактивних програм.

Потрібно відмітити, що правильне визначення алгоритмів планування процесів сильно впливає на продуктивність усієї системи.

Головними причинами, що призводять до зменшення продуктивності системи є:

- накладні витрати на переключення процесора (сюди відносять переключення контекстів, переміщення сторінок віртуальної пам'яті, заміну даних у КЕШі і т.п.);
- переключення на інший процес, якщо попередній виконує критичну ділянку. У цьому випадку, втрати є найбільшими.

Використання динамічних пріоритетів

При роботі у режимі реального часу, коли виконуються задачі контролю та управління, може виникнути ситуація, коли одна або декілька задач не можуть бути виконані протягом досить великого проміжку часу. Втрати, пов'язані з невиконанням таких задач, можуть бути досить значними. При цьому доцільно тимчасово змінити пріоритети „аварійних" задач. Після їх

виконання відповідні пріоритети можна поновити. Тому, як правило, у ОС реального часу є засоби зміни пріоритету програми. Одним з таких засобів є наступний.

При обчисленні беруть участь два поля дескриптора процесу. Перше з них визначається користувачем явно або формується за замовчуванням за допомогою системи програмування. Друге поле формується планувальником і зветься системною складовою або поточним пріоритетом. Тобто процес має два атрибути пріоритету: поточний пріоритет, відповідно до якого здійснюється планування, та заказний відносний пріоритет. Наприклад, більш високому значенню поточного пріоритету може відповідати більш низький фактичний пріоритет планувальника.

Розподіл між пріоритетами режиму ядра та задачі також залежить від версії ОС. Пріоритети процесів, що виконуються у режимі задачі, мають менший пріоритет ніж у режимі ядра. Наприклад, пріоритети режиму задачі змінюються у діапазоні 0-65, для режиму ядра – 66-95, а процеси, пріоритети яких лежать у діапазоні 96-127 – процеси з фіксованим пріоритетом. Вони не змінюються ОС та визначені для підтримки реального часу.

Як правило, процесу, що чекає недоступного у даний час ресурсу, система визначає пріоритет, який вибирається ядром із діапазону системних пріоритетів. Коли процес завершується, ядро встановлює значення поточного пріоритету, що дорівнює пріоритету сну. Оскільки пріоритет такого процесу знаходиться у системному діапазоні та вищий, ніж пріоритет задачі, ймовірність надання процесу обчислювальних ресурсів значно збільшується.

Скажімо, такий підхід дозволяє швидко завершити системний виклик, виконання якого можуть блокувати деякі системні ресурси.

Після завершення системного виклику та перед поверненням у режим задачі ядро поновить пріоритет режиму задачі, що зберігся перед виконанням системного виклику. Це може призвести до зниження пріоритету, що, у свою чергу, може викликати переключення контексту.

Наприклад, операційна система OS/2 самостійно змінює пріоритет виконання програми незалежно від рівня, який був встановлений прикладною задачею. Цей механізм зветься механізмом підвищення пріоритету.

ОС змінює пріоритети задачі у трьох випадках:

- збільшується пріоритет активної задачі. Пріоритет автоматично збільшується, коли вона стає активною. Це знижує час реакції активного додатку на дії користувача у порівнянні із фоновими програмами;
- збільшується пріоритет введення-виведення. По завершенні операцій введення-виведення процес отримує найбільший рівень пріоритету його класу. Таким чином забезпечується завершення усіх незакінчених операцій введення-виведення;
- збільшення пріоритету „забутих" задач. Якщо задача (або процес) не отримує процесор досить довго (цей проміжок часу задається наприклад у OS/2 у файлі CONFIG.SYS), диспетчер задач OS/2 тимчасово присвоює їй рівень пріоритету, що перевищує критичний. У результаті, час переключення на таку „забуту" програму зменшується. Після використання одного кванту часу пріоритет знову зменшується до попереднього.

Такий механізм у сильно завантажених системах дозволяє програмам з низьким пріоритетом теж працювати. При відсутності механізму, такі програми взагалі б не працювали.

4.4 Контрольні запитання до розділу 4

1.Дайте визначення терміну мультипрограмування.

2.Яким чином досягається сполучення операцій обчислення і введення-виведення?

3. Дайте визначення термінам дескриптор процесу і контекст процесу.

4. Яким чином відбувається перехід від виконання одного потоку до іншого?

5. Які завдання вирішуються при плануванні процесів?

6. Яка різниця між динамічним і статичним плануванням?

7 У чому полягає процес диспетчеризації?

8. Яка різниця між витісняючими і невитісняючими алгоритмами планування?

9. Які особливості алгоритмів планування, заснованих на квантуванні?

10. Які особливості алгоритмів планування, заснованих на пріоритетах?

11. Які особливості планування у системах реального часу?

12. Дайте визначення терміну багаторівнева черга.

13. Які особливості використання динамічних пріоритетів?

Розділ 5. Мультипрограмування на основі переривань

5.1 Типи переривань

Переривання відбувається в довільній точці потоку команд програми, що програміст не може прогнозувати.

Деяка подібність переривань із процедурою є в тому, що в обох випадках виконується підпрограма, що обробляє цю ситуацію, а потім продовжується виконання основної гілки програми.

Залежно від джерела, переривання поділяють на три великі класи:

- зовнішні;
- внутрішні;
- програмні.

Зовнішні переривання виникають у результаті дій користувача або в результаті надходження сигналів від апаратних пристроїв. Даний клас переривань є **асинхронним** стосовно потоку інструкцій і перериває програми.

Апаратура процесора працює так, що асинхронні переривання виникають між виконанням двох сусідніх інструкцій, при цьому система після обробки переривання продовжує виконання процесу, починаючи вже з наступної інструкції.

Внутрішні переривання відбуваються **синхронно** виконанню програми при появі аварійної ситуації в ході виконання деякої інструкції програми. Прикладами є ділення на нуль, помилки захисту пам'яті, звертання до неіснуючої адреси. Переривання виникають усередині виконання команди.

Програмні переривання відрізняються від попередніх двох класів тим, що вони, по своїй суті, не є «справжніми» перериваннями. Програмне переривання виникає при виконанні спеціальної команди процесора. Її

виконання імітує переривання, тобто перехід на нову послідовність інструкцій.

Перериванням приписується пріоритет, за допомогою якого вони ранжуються за ступенню важливості й терміновості. Про переривання, що мають однакове значення пріоритету, говорять, що вони відносяться до **одного рівня пріоритету** переривань.

Процедури, що викликаються за перериваннями, звичайно називають **оброблювачами переривань** або **процедурами обслуговування переривань**.

Апаратні переривання обробляються драйверами відповідних зовнішніх пристроїв, внутрішні переривання – спеціальними модулями ядра, а програмні переривання – процедурами ОС, що обслуговують системні виклики.

Крім цих модулів, в ОС може містити так званий диспетчер переривань, що координує роботу окремих оброблювачів переривань.

Узагальнено, послідовність дій апаратних і програмних засобів по обробці переривання можна описати таким чином:

1. При виникненні сигналу (для апаратних переривань) або умови (для внутрішніх переривань) переривання відбувається первинне апаратне розпізнавання типу переривання. Якщо переривання даного типу в даний момент заборонені (пріоритетною схемою або механізмом маскування), то процесор продовжує підтримувати природний хід виконання команд.

   У противному випадку, залежно від інформації, що надійшла в процесор, відбувається автоматичний виклик процедури обробки переривання, адреса якої знаходиться у спеціальній таблиці ОС, розташованій або в регістрах процесора, або в певному місці оперативної пам'яті.
2. Автоматично зберігається деяка частина контексту перерваного потоку, що дозволить ядру відновити виконання потоку процесу після обробки переривання. У цю підмножину включають значення лічильника команд, слова стану машини, що зберігає ознаки основних режимів роботи процесора (наприклад, слова – регістр EFLAGS в Intel

Pentium), а також декількох регістрів загального призначення, які потрібні програмі обробки переривання. Може бути збережений і повний контекст процесу, якщо ОС обслуговує дане переривання зі зміною процесу.

3. Одночасно із завантаженням адреси процедури обробки переривань у лічильник команд може автоматично виконуватися завантаження нового значення слова стану машини, що визначає режими роботи процесора при обробці переривання, у тому числі роботу в привілейованому режимі.
4. Тимчасово забороняються переривання даного типу, щоб не утворилася черга вкладених друг у друга потоків однієї й тієї ж процедури. Деталі виконання цієї операції залежать від особливостей апаратної платформи, наприклад, може використовуватися механізм маскування переривань.
5. Після того, як переривання оброблене ядром ОС, перерваний контекст відновлюється, і робота потоку відновлюється з перерваного місця.

Частина контексту відновлюється апаратно по команді повернення з переривань (наприклад, адреса наступної команди й слово стану машини), а частина – програмним способом, за допомогою явних команд читання даних зі стеку. При поверненні з переривання блокування повторних переривань даного типу знімається.

Програмне переривання реалізує один зі способів переходу на підпрограму за допомогою спеціальної інструкції процесора, такої як INT у процесорах Intel Pentium, trap у процесорах Motorola і т.п.

При виконанні команди програмного переривання процесор відпрацьовує ту ж послідовність дій, що й при виникненні зовнішнього або внутрішнього переривання, проте це відбувається в передбачуваній точці програми — там, де програміст помістив дану команду.

Практично всі сучасні процесори мають у системі команд інструкції програмних переривань.

Програмні переривання часто використовуються для виконання обмеженої кількості викликів функцій ядра операційної системи, тобто системних викликів.

5.2 Диспетчеризація і пріоритезація переривань в ОС

Переривання виконують корисну для обчислювальної системи функцію – дозволяють реагувати на асинхронні стосовно обчислювального процесу події.

Переривання створюють додаткові труднощі для ОС в організації обчислювального процесу.

Для впорядкування роботи оброблювачів переривань в ОС застосовується той же механізм, що й для впорядкування роботи користувацьких процесів – механізм пріоритетних черг. Всі джерела переривань поділяють на декілька класів, причому кожному класу присвоюються свій пріоритет. В ОС виділяється програмний модуль, що займається диспетчеризацією оброблювачів переривань. Цей модуль у різних ОС називається по-різному. Визначимо його як **диспетчер переривань**.

При виникненні переривання диспетчер переривань викликається першим. Він забороняє на якийсь час всі переривання і з'ясовує причину переривання. Потім порівнюється пріоритет джерела переривання з поточним пріоритетом потоку команд, виконуваних процесором. Якщо пріоритет нового запиту вище поточного, то виконання поточного потоку припиняється, й виконується обробка переривання. У противному випадку, запит ставиться в чергу.

5.3 Процедури обробки переривань і поточний процес

Важливою особливістю процедур, виконуваних за запитами переривань, є те, що вони виконують роботу, найчастіше ніяк не пов'язану з поточним процесом.

Наприклад, драйвер диску може одержати керування після того, як контролер диску записав у відповідні сектори інформацію, отриману від процесу А, але цей момент часу, не збігається з періодом чергової ітерації виконання процесу А або його потоку.

У найбільш типовому випадку, процес А перебуватиме в стані очікування завершення операції введення-виведення (при синхронному режимі виконання цієї операції), і драйвер диску перерве який-небудь інший процес.

У деяких випадках взагалі важко однозначно визначити, для якого процесу виконує роботу той або інший програмний модуль ОС, наприклад, планувальник потоків. Тому для такого роду процедур вводяться обмеження – вони не мають права використовувати ресурси (пам'ять, відкриті файли тощо), з якими працює поточний процес.

Процедури обробки переривань працюють із ресурсами, які були виділені їм при ініціалізації відповідного драйвера або ініціалізації самої операційної системи. Ці ресурси належать ОС, а не конкретному процесу. Так пам'ять драйверам виділяється із системної області. Тому звичайно говорять, що процедури обробки переривань працюють поза контекстом процесу.

Диспетчеризація переривань є важливою функцією ОС, і ця функція реалізована практично у всіх мультипрограмних ОС. Як правило, в ОС реалізується дворівневий механізм планування робіт. Верхній рівень планування виконується диспетчером переривань, що розподіляє процесорний час між потоком вхідних запитів на переривання різних типів – зовнішніх, внутрішніх і програмних. Процесорний час, що залишився, розподіляється іншим диспетчером – диспетчером потоків на підставі дисциплін квантування й інших, раніше розглянутих дисциплін.

5.4 Системні виклики

Системний виклик дозволяє додатку звернутися до ОС із проханням виконати ту чи іншу дію, оформлену як процедура (або набір процедур) кодового сегменту ОС.

Для прикладного програміста ОС виглядає як деяка бібліотека, що реалізує корисні функції, які полегшують керування прикладним завданням або виконання дій, заборонених у користувацькому режимі, наприклад обмін даними із пристроєм введення-виведення.

Реалізація системних викликів повинна задовольняти наступним вимогам:

- забезпечувати перемикання в привілейований режим;
- мати високу швидкість виклику процедур ОС;
- забезпечувати однакове звертання до системних викликів для всіх апаратних платформ, на яких працюють ОС;
- допускати просте розширення набору системних викликів;
- забезпечувати контроль з боку ОС за коректним використанням системних викликів.

У більшості ОС системні виклики обслуговуються за централізованою схемою, що базується на існуванні диспетчера системних викликів (рис. 5.1).

При будь-якому системному виклику додаток виконує програмне переривання з певним та єдиним номером вектора.

Перед виконанням програмного переривання додаток передає ОС номер системного виклику. Спосіб передачі залежить від реалізації. Наприклад, номер можна помістити в певний регістр процесора або передати через стек. Також деяким способом передаються аргументи системного виклику, вони можуть міститися як, у регістрах загального призначення, так і передаватися через стек або масив в оперативній пам'яті.

Після завершення роботи системного виклику керування повертається диспетчеру, який одержує також код завершення цього виклику. Диспетчер відновлює регістри процесора, поміщає в певний регістр код повернення й виконує інструкцію повернення з переривання, що відновлює непривілейований режим роботи процесора.

Описаний табличний спосіб організації системних викликів прийнятий практично у всіх операційних системах. Він дозволяє легко модифікувати

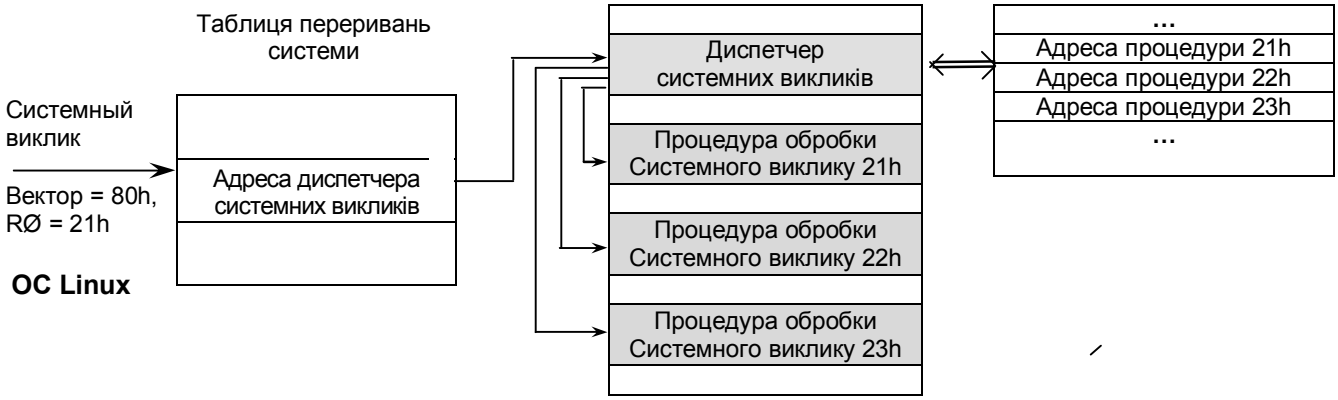
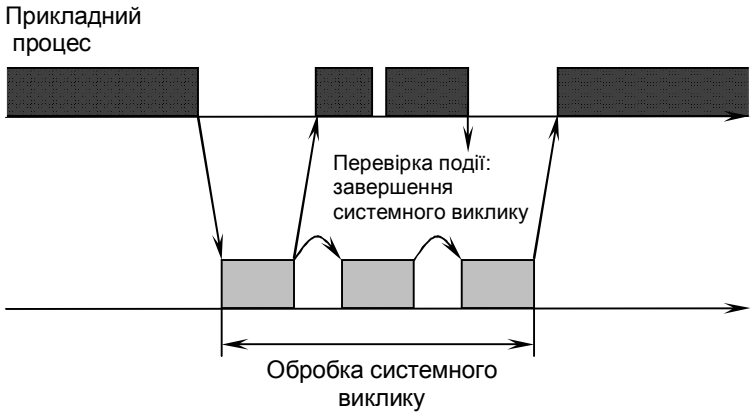


Рис. 5.1 Централізована схема обробки системних викликів

набір системних викликів, просто додавши в таблицю нову адресу й розширюючи діапазон припустимих номерів викликів.

ОС може виконувати системні виклики в **синхронному** або **асинхронному** режимах (рис. 5.2).

а) Асинхронний системний виклик



б) Синхронний системний виклик

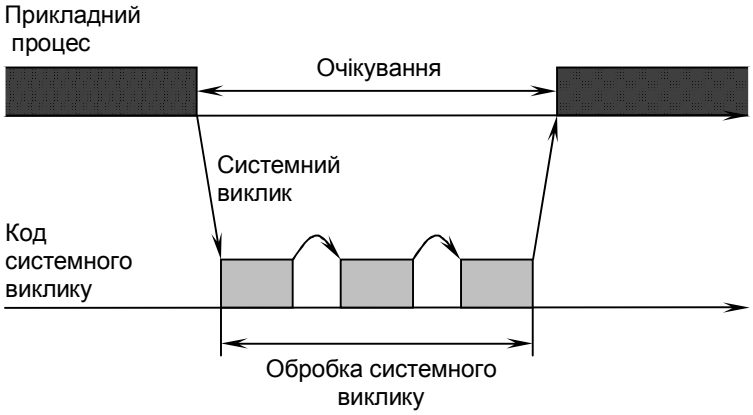


Рис.5.2. Системні виклики

Синхронний системний виклик означає, що процес, який зробив такий виклик, призупиняється, доки системний виклик не виконає всю необхідну роботу. Після цього планувальник переводить процес у стан готовності.

Асинхронний системний виклик не призводить до переводу процесу в режим очікування, і після виконання деяких початкових системних дій, наприклад, запуску операції виводу-виводу, керування повертається прикладному процесу. Більшість системних викликів в ОС є синхронними.

5.5 Контрольні запитання до розділу 5

1. На які класи поділяють переривання?
2. Як називають процедури, які виконуються за перериваннями?
3. Дайте визначення програмним перериванням.
4. Дайте визначення системному виклику.
5. Яким вимогам повинна задовольняти реалізація системного виклику?
6. Які особливості виконання синхронного і асинхронного системних викликів?

Розділ 6. Керування пам'яттю

6.1 Функції ОС по керуванню пам'яттю

Під **пам'яттю** розуміють оперативну пам'ять комп'ютера. На відміну від пам'яті жорсткого диска, що називають **зовнішньою пам'яттю**, оперативна пам'ять для збереження інформації вимагає постійного електроживлення.

Особлива роль пам'яті полягає в тому, що процесор може виконувати інструкції програми тільки в тому випадку, якщо вони перебувають у пам'яті.

Пам'ять розподіляється як між модулями прикладних програм, так і між модулями самої операційної системи.

Функціями ОС по керуванню пам'яттю в мультипрограмній системі є:

- відстеження вільної й зайнятої пам'яті;
- виділення пам'яті процесам і звільнення пам'яті по завершенні процесів;
- витіснення кодів і даних процесів з оперативної пам'яті на диск, коли розміру основної пам'яті недостатньо для розміщення в ній всіх процесів, і повернення їх в оперативну пам'ять, коли в ній звільняється місце;
- настроювання адрес програми на конкретну область фізичної пам'яті.

Під час роботи ОС доводиться створювати нові службові інформаційні структури, такі як описувачі процесів і потоків, різні таблиці розподілу ресурсів, буфери для обміну даними й т.п. Всі ці системні об'єкти вимагають пам'яті.

У деяких ОС під час установки резервується деякий фіксований обсяг пам'яті для системних потреб. В інших же ОС використовується більш гнучкий підхід, при якому пам'ять для системних цілей виділяється динамічно.

Захист пам'яті — ще одне важливе завдання ОС. Воно полягає в тому, щоб не дозволити виконуваному процесу записувати або читати дані з пам'яті, призначеної іншому процесу.

6.2 Типи адрес. Завантаження процесу

Для ідентифікації змінних і команд на різних етапах життєвого циклу програми використовуються символьні імена, віртуальні адреси й фізичні адреси (рис. 6.1):

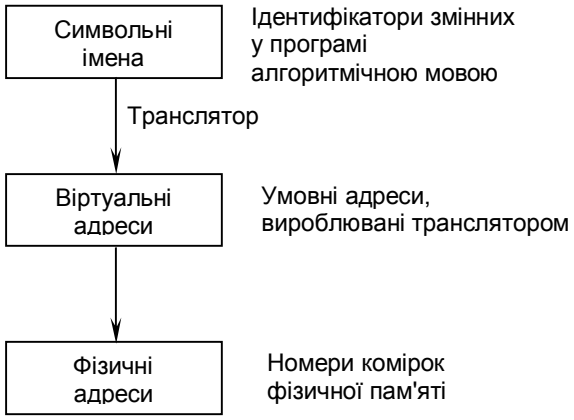


Рис. 6.1. Типи адрес

- **Символьні імена** надає користувач при написанні програми.
- **Віртуальні (умовні) адреси**, виробляє транслятор, що перетворює програму на машинну мову.
- **Фізичні адреси** відповідають номерам комірок оперативної пам'яті, де в дійсності розташовані змінні й команди.

Сукупність віртуальних адрес називають **віртуальним адресним простором**.

Діапазон можливих адрес віртуального простору у всіх процесів є однаковим. Проте, кожен процес має власний віртуальний адресний простір - транслятор привласнює віртуальні адреси змінним і кодам кожної програмі незалежно.

У різних ОС використовуються різні способи структуризації адресного простору:

1. ***Лінійна*** послідовність віртуальних адрес. Таку структуру адресного простору називають також **пласкою (flat***)*. При цьому віртуальною адресою є єдине число, що представляє собою зсув відносно початку віртуального адресного простору. Адресу такого типу називають лінійною віртуальною адресою.
2. Віртуальний адресний простір поділяють на частині, називані сегментами. Віртуальна адресао представляє собою **пари** чисел (**n**, **m**)*,* де **n** визначає сегмент, а **m** — зсув усередині сегмента.
3. Є більш складні способи структуризації, коли віртуальна адреса утворюється трьома (або навіть більше) числами.

Існують два принципово різних підходи до перетворення віртуальних адрес у фізичні.

1. Заміна віртуальних адрес на фізичні виконується один раз для кожного процесу під час початкового завантаження програми у пам'ять. Виконує це системна програма — *переміщуючий* **завантажувач.** На підставі наявних у неї вихідних даних про початкову адресу фізичної пам'яті, а також інформації, наданої транслятором про адресно-залежні елементи програми, він виконує завантаження програми, поєднуючи із заміною віртуальних адрес фізичними.
2. Програма завантажується у пам'ять в незміненому вигляді у віртуальних адресах. При завантаженні ОС фіксує зсув дійсного розташування програмного коду щодо віртуального адресного простору. Під час виконання програми при кожному звертанні до оперативної пам'яті виконується перетворення віртуальної адреси у фізичну.

Останній спосіб є більш гнучким: у той час як переміщуючий завантажувач жорстко прив'язує програму до початку виділеної їй ділянки пам'яті, динамічне перетворення віртуальних адрес дозволяє переміщати програмний код процесу протягом усього періоду його виконання.

Як правило, обсяг віртуального адресного простору перевищує доступний обсяг оперативної пам'яті. У такому випадку, для зберігання даних

віртуального адресного простору ОС використовує програмну зовнішню пам'ять. Однак, співвідношення обсягів віртуальної й фізичної пам'яті може бути й зворотним.

Варто пам'ятати, що, у загальному випадку, механізми віртуального адресного простору і віртуальної пам'яті – це не одне й теж саме. Можна уявити собі ОС, у якій підтримуються віртуальні адресні простори для процесів, але відсутній механізм віртуальної пам'яті. Це можливо тільки у тому випадку, якщо розмір віртуального адресного простору кожного процесу менше обсягу фізичної пам'яті.

Вміст призначеного процесу віртуального адресного простору являє собою **образ процесу**.

Під час роботи процесу постійно виконуються переходи від прикладних кодів до кодів ОС, які або явно викликаються із прикладних процесів як системні функції, або викликаються як реакція на зовнішні події. Для того щоб спростити передачу керування від прикладного коду до коду ОС, а також для простого доступу модулів ОС до прикладних даних, у більшості ОС її сегменти розділяють віртуальний адресний простір із прикладними сегментами активного процесу.

Тобто, віртуальний адресний простір процесу ділиться на дві неперервні частини: системну й користувацьку. У деяких ОС (наприклад, Windows NT), ці частини мають однаковий розмір – по 2 Гбайти.

Частина віртуального адресного простору кожного процесу, що відводять під сегменти ОС, є ідентичною для всіх процесів. Тому при зміні активного процесу заміняється тільки друга частина віртуального адресного простору. Наприклад, у процесорах Intel Pentium існують два типи системних таблиць: одна – для опису сегментів, загальних для всіх процесів, інша – для опису індивідуальних сегментів даного процесу. При зміні процесу перша таблиця залишається незмінною, а друга – замінюється новою.

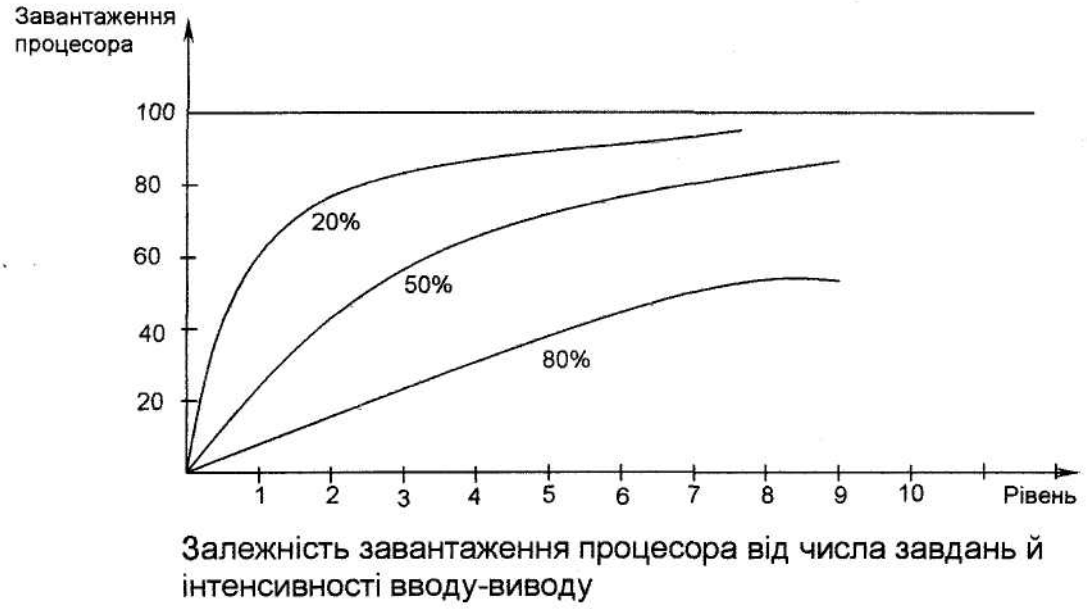


Рис. 6.2. Завантаження процесора

Велика кількість завдань вимагає великих обсягів оперативної пам'яті.

В умовах, коли для забезпечення прийнятного рівня мультипрограмування наявної пам'яті виявляється недостатньо, був запропонований метод організації обчислювального процесу, при якому образи деяких процесів цілком або частково тимчасово вивантажуються на диск.

У мультипрограмному режимі крім активного процесу є також призупинені процеси, що очікують завершення введення-виведення або звільнення ресурсів, а також процеси у стані готовності, що стоять у черзі до процесора.

Образи таких неактивних процесів можуть бути тимчасово, до наступного циклу активності, вивантажені на диск. Незважаючи на це, ОС «знає» про існування процесів і враховує це при розподілі процесорного часу й інших ресурсів. До моменту, коли надходить черга виконання вивантаженого процесу, його образ повертається з диска в оперативну пам'ять. Якщо при цьому виявляється, що вільного місця в оперативній пам'яті не вистачає, то на диск вивантажується інший процес. Така підміна

(віртуалізація) оперативної пам'яті дисковою дозволяє підвищити рівень мультипрограмування.

Віртуальним називають ресурс, який представляється користувачу або користувальницькій програмі таким, що володіє властивостями, якими він насправді не володіє. Ясно, що робота віртуальної оперативної пам'яті відбувається повільніше, ніж реальної.

Віртуалізація оперативної пам'яті здійснюється сукупністю програмних модулів ОС й апаратних схем процесора і включає рішення наступних завдань:

- розміщення даних у запам'ятовуючих пристроях різного типу;
- вибір образів процесів або їхніх частин для переміщення з оперативної пам'яті на диск і зворотно;
- переміщення в міру необхідності даних між пам'яттю й диском;
- перетворення віртуальних адрес у фізичні.

Зауважимо, що з проблемою розміщення в оперативній пам'яті програм, розміри яких перевищують розміри оперативної пам'яті, програмісти зіштовхнулися досить давно. Одним із рішень було розбиття програми на частини, які називали **оверлеями.** Коли один оверлей закінчував своє виконання, він викликав інший оверлей. Всі оверлеї зберігалися на диску й переміщувалися між пам'яттю й диском засобами ОС на підставі директив програміста, що містилися у програмі.

Незважаючи на зовнішню подібність, ця процедура має принципову відмінність від віртуальної пам'яті. Вона полягає в тому, що при оверлеї розбиття програми на частини й планування їхнього завантаження в оперативну пам'ять виконуються програмістом заздалегідь під час написання програми.

Віртуалізація пам'яті може бути здійснена на основі двох різних підходів:

- **свопінг** — образи процесів вивантажуються на диск і повертаються в оперативну пам'ять *повністю;*

- **віртуальна пам'ять** — між оперативною пам'яттю й диском переміщаються *частини* (сегменти, сторінки) образів процесів.

Свопінг – окремий випадок віртуалізації. Це найбільш простий спосіб спільного використання оперативної пам'яті й диску.

Недолік – надмірність. Коли ОС вирішує активізувати процес, для його виконання часто не потрібно завантажувати в оперативну пам'ять всі його сегменти повністю. Аналогічно, при звільненні пам'яті для завантаження нового процесу дуже часто не потрібно повністю вивантажувати інший процес на диск. Досить вивантажити тільки частину його образу. Переміщення надлишкової інформації сповільнює роботу системи, й при цьому неефективно використовується оперативна пам'ять.

Ще один *недолік.* Системи, що підтримують стопінг, не здатні завантажити для виконання процес, віртуальний адресний простір якого перевищує наявну вільну пам'ять. У сучасних ОС свопінг застосування не знаходить.

Ключовою проблемою віртуальної пам'яті, що виникає в результаті багаторазової зміни місця розташування в оперативній пам'яті образів процесів або їхніх частин, є *перетворення **віртуальних** адрес у фізичні.*

У цей час вся безліч реалізацій віртуальної пам'яті може бути представлена трьома наступними класами:

- **Сторінкова віртуальна пам'ять.** Переміщення даних здійснюється сторінками – частинами віртуального адресного простору, фіксованого й порівняно невеликого розміру.
- **Сегментна віртуальна пам'ять. П**ереміщення здійснюється сегментами — частинами віртуального адресного простору довільного розміру.
- **Сегментно-сторінкова віртуальна пам'ять. В**икористовується дворівневий поділ: віртуальний адресний простір ділять на сегменти, а потім сегменти ділять на сторінки. Одиницею переміщення тут є сторінка.

Для тимчасового зберігання сегментів і сторінок на диску виділяють або спеціальну ділянку, або спеціальний файл, що називають сторінковим файлом. Поточний розмір сторінкового файлу є важливим параметром ОС – чим він більше, тим більше програм ОС може одночасно виконувати. Однак, зі збільшенням сторінкового файлу збільшується час на перекачування інформації, й загальна корисна продуктивність системи зменшується.

Розмір сторінкового файлу в сучасних ОС є параметром, що настроюється.

6.3 Алгоритми розподілу пам'яті

Алгоритми розподілу пам'яті ділять на два класи: алгоритми, у яких використовується переміщення сегментів процесів між оперативною пам'яттю й диском, і алгоритми, у яких зовнішня пам'ять не залучається (рис. 6.3).

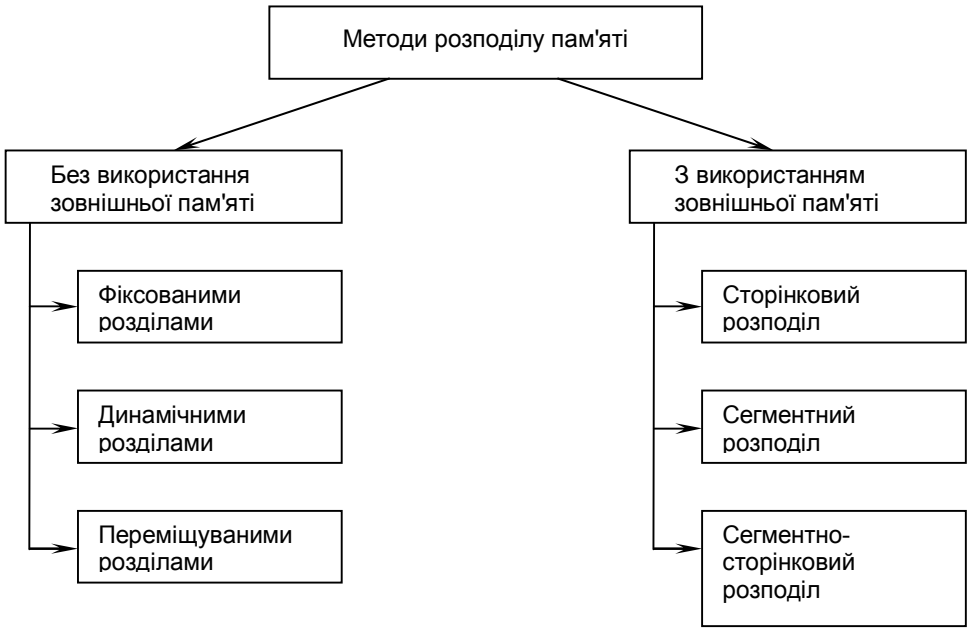


Рис. 6.3. Класифікація методів розподілу пам’яті

Розподіл пам'яті фіксованими розділами

Це найпростіший спосіб керування пам'яттю. Пам'ять розбивається на кілька областей фіксованої величини, називаних *розділами.*

Це розбиття може бути виконано вручну оператором під час старту системи або її установки. Після цього границі розділів не змінюються.

Новий процес, що надійшов на виконання, міститься або в загальну чергу, або в чергу до деякого розділу.

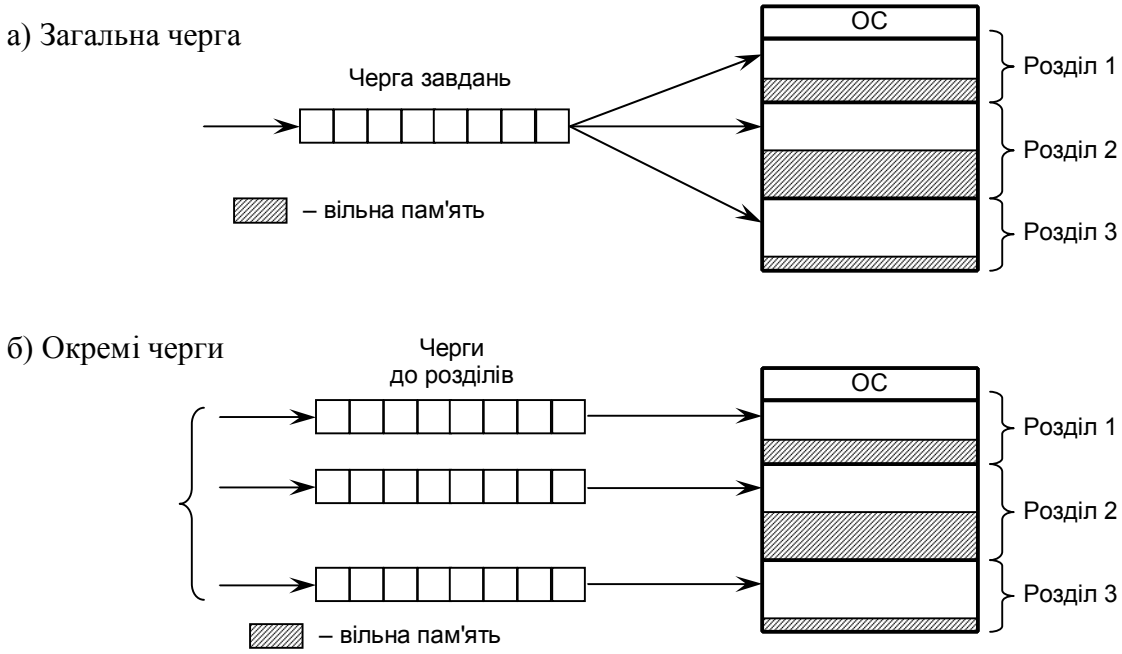


Рис. 6.4. Розподіл пам'яті фіксованими розділами

Система керування пам'яті вирішує наступні завдання:

- Порівнює обсяг пам'яті, необхідний для нового процесу, з розмірами вільних розділів і вибирає підходящий розділ.
- Здійснює завантаження програми в один з розділів і настроювання адрес.

Уже на етапі трансляції розроблювач програми може задати розділ, у якому її варто виконувати. Це дозволяє відразу, без використання переміщуючого завантажувача, одержати машинний код, настроєний на конкретну область пам'яті.

Істотний недолік – жорсткість.

Рівень мультипрограмування заздалегідь обмежений числом розділів. Незалежно від розміру програми, вона буде займати весь розділ; з іншого боку, процес, що вимагає кілька розділів, не може бути виконаний.

Цей найпростіший метод розподілу пам'яті зараз знаходить застосування тільки в системах реального часу, завдяки детермінованості обчислювального процесу.

Розподіл пам'яті динамічними розділами

Кожному новому процесу, що надходить на виконання, виділяється вся необхідна йому пам'ять; якщо її не вистачає, процес не запускається.

Для реалізації методу потрібні наступні функції ОС:

- Ведення таблиць вільних і зайнятих областей, у яких вказуються початкові адреси й розміри ділянок пам'яті.
- При створенні нового процесу – аналіз вимог до пам'яті, перегляд таблиці вільних областей і вибір розділу.

  Вибір розділу може здійснюватися за різними правилами: перший знайдений розділ достатнього розміру; розділ з найменшим достатнім розміром; розділ з найбільшим достатнім розміром.
- Завантаження програми у виділений їй розділ і коригування таблиць вільних і зайнятих областей.
- Коригування таблиць вільних і зайнятих областей.

Цьому методу властивий серйозний недолік – фрагментація пам'яті.

Фрагментація – наявність великої кількості несуміжних ділянок вільної пам'яті маленького розміру. Такого, що жодна з програм, що надходять на виконання, не може поміститися в жодній з ділянок, хоча сумарний обсяг вільних фрагментів може скласти величину, що перевищує необхідний програмі обсяг пам'яті.

Прикладом застосування цього методу є популярна в минулому OS/360 й ЕС ЕОМ.

Переміщувані розділи

Боротьба із фрагментацією здійснюється шляхом переміщення всіх зайнятих ділянок убік старших або молодших адрес таким чином, щоб вся вільна пам'ять утворила єдину вільну область (рис. 6.5).

На додаток до функцій, які виконує ОС із динамічними розділами, вона ще час від часу копіює вміст розділів з одного місця пам'яті в інше, коригуючи таблиці вільних і зайнятих областей. Цю процедура називають **стискуванням.**

Стискування виконується або при завершенні кожного процесу, або у випадку відсутності для створюваного процесу вільного розділу достатнього розміру.

Оскільки програми переміщюються в оперативній пам'яті в процесі свого виконання, то, у цьому випадку, неможливо виконати настроювання адрес за допомогою переміщуючого завантажувача. Тут необхідна реалізація динамічного перетворення адрес.

Хоча цей метод і призводить до більш ефективного використання пам'яті, він вимагає значних часових витрат, що нівелює переваги даного методу.

Метод використовувся в ранніх версіях ОС OS/2.

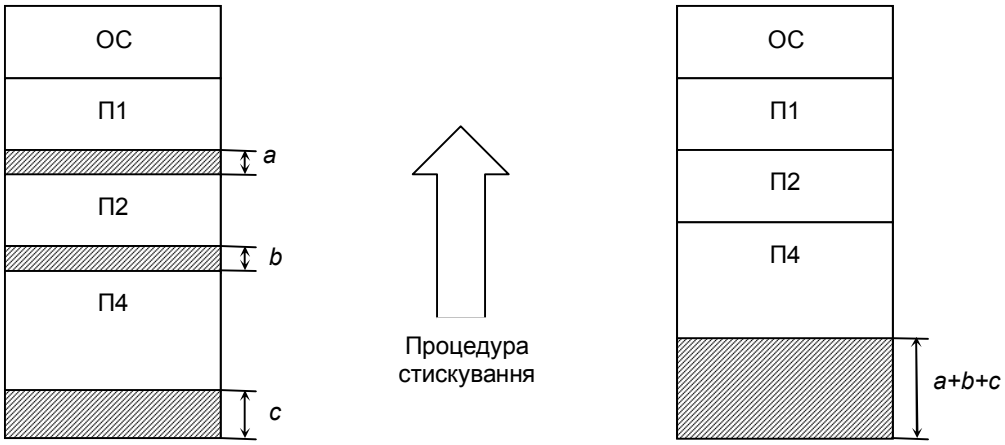


Рис. 6.5. Розподіл пам'яті переміщуваними розділами

Свопінг і віртуальна пам'ять. Обсяг оперативної пам'яті, що є в комп'ютері, істотно позначається на протіканні обчислювального процесу. Він обмежує кількість одночасно виконуваних програм і розміри їх віртуальних адресних просторів.

Якщо всі завдання мультипрограмної суміші є обчислювальними (мало операцій введення-виведення), для оптимального завантаження процесора може виявитися достатнім усього 3-5 завдань.

Якщо обчислювальна система завантажена виконанням інтерактивних завдань, для ефективного використання процесора може знадобитися вже кілька десятків, а то й сотень завдань.

Сторінковий розподіл

Віртуальний адресний простір кожного процесу ділиться на частини однакового, фіксованого для даної системи розміру – **віртуальні сторінки** (рис. 6.6).

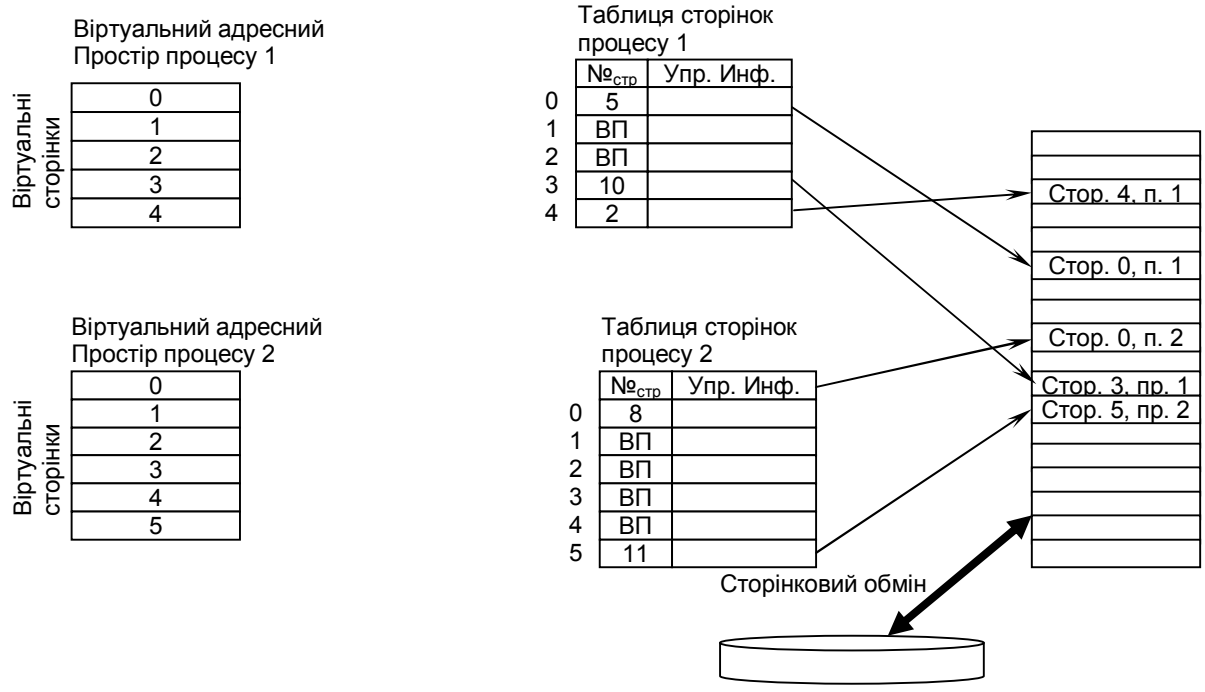


Рис. 6.6. Сторінковий розподіл

У загальному випадку, розмір віртуального адресного простору процесу не кратний розміру сторінки. Тому остання сторінка кожного процесу доповнюється фіктивною областю.

Вся оперативна пам'ять машини також ділиться на частині такого ж розміру – **фізичні сторінки**. Розмір сторінки вибирається рівним степеням двійки: 512, 1024, 4096 байт і т.д, Це дозволяє спростити механізм перетворення адрес.

Для кожного процесу ОС створює **таблицю сторінок** — інформаційну структуру, що містить записи про всі віртуальні сторінки процесу.

Запис таблиці має назву **дескриптора сторінки.** Він містить наступну інформацію:

- номер фізичної сторінки, у яку завантажена дана віртуальна сторінка;
- ознака присутності; установлюється в одиницю, якщо віртуальна сторінка перебуває в оперативній пам'яті;
- ознака модифікації сторінки, що встановлюється в одиницю щоразу, коли здійснюється запис за адресою, що відповідає даній сторінці;
- ознака звертання до сторінки, називана також бітом доступу, що встановлюється в одиницю при кожному звертанні за адресою, що відповідає даній сторінці.

Ознаки присутності, модифікації й звертання у більшості сучасних процесорів встановлюються апаратно при операціях з пам'яттю. Інформація з таблиць сторінок використовується для вирішення питання про необхідність переміщення тієї чи іншої сторінки між пам'яттю й диском, а також для перетворення віртуальної адреси у фізичну.

Таблиці сторінок розміщують в оперативній пам'яті. Адреса таблиці сторінок включається в контекст відповідного процесу. При активізації чергового процесу ОС завантажує адресу його таблиці сторінок у спеціальний регістр процесора.

При кожному звертанні до пам'яті виконується пошук номера віртуальної сторінки, що містить необхідну адресу, потім за цим номером визначається

потрібний елемент таблиці сторінок, і з нього читається інформація, що описує сторінку. Далі аналізується ознака присутності, і, якщо сторінка перебуває в оперативній пам'яті, виконується перетворення віртуальної адреси у фізичну. Якщо ж потрібна віртуальна сторінка в цей момент вивантажена на диск, то відбувається **сторінкове переривання**.

Процес, що виконується, переводиться в стан очікування, і активізується інший процес із черги процесів, що перебувають у стані готовності. Паралельно програма обробки сторінкового переривання знаходить на диску необхідну віртуальну сторінку й намагається завантажити її в оперативну пам'ять. Якщо в пам'яті є вільна фізична сторінка, то завантаження виконується негайно, якщо ж вільних сторінок немає, то, на підставі прийнятої в даній системі стратегії заміщення сторінок, вирішується питання про те, яку саме сторінку варто вивантажити з оперативної пам'яті.

Після визначення віртуальної сторінки, присвоюється нуль її біту присутності й аналізується її ознака модифікації. Якщо за час останнього перебування в оперативній пам'яті вона була модифікована, її нова версія повинна бути переписана на диск. Якщо ні, то ніякого запису на диск не робиться (вона й так є). Фізична сторінка оголошується вільною.

Віртуальна адреса при сторінковому розподілі може бути представлена у вигляді пари (**p, s_v**), де **p** — порядковий номер віртуальної сторінки процесу (нумерація сторінок починається з 0), а **s_v** — зсув у межах віртуальної сторінки.

Фізична адреса також може бути представлена у вигляді пари (**n, S_f**), де **n** — номер фізичної сторінки, а **S_f** — зсув у межах фізичної сторінки. Завдання підсистеми віртуальної пам'яті полягає у перетворенні пари (**p, s_v**) в (**n, S_f**).

Розглянемо дві базисних властивості сторінкової організації.

Перша з них полягає в тому, що обсяг сторінки обирається рівним степені двійки — $\mathbf{2^k}$. Із цього випливає, що зсув **s** може бути отримано простим відділенням **k** молодших розрядів у двійковому записі адреси, а решта старших розрядів адреси являє собою двійковий запис номера сторінки.

Наприклад:

Розмір сторінки 1 Кбайт (2^{10}). Двійкова адреса **101 000 111 001$_2$**. Вона належить сторінці 10_2 і зміщена відносно її початку на 1 000 111 001_2 байт.

Друга властивість полягає в тому, що в межах сторінки неперервна послідовність віртуальних адрес однозначно відображається у неперервну послідовність фізичних адрес, тобто $s_v = S_f$.

Звідси випливає проста схема перетворення віртуальних адрес у фізичні.

Молодші розряди фізичної адреси, що відповідають зсуву, отримують перенесенням такої ж кількості молодших розрядів з віртуальної адреси. Старші розряди фізичної адреси, що відповідають номеру фізичної сторінки, визначаються з таблиці сторінок, в якій вказується відповідність віртуальних і фізичних сторінок.

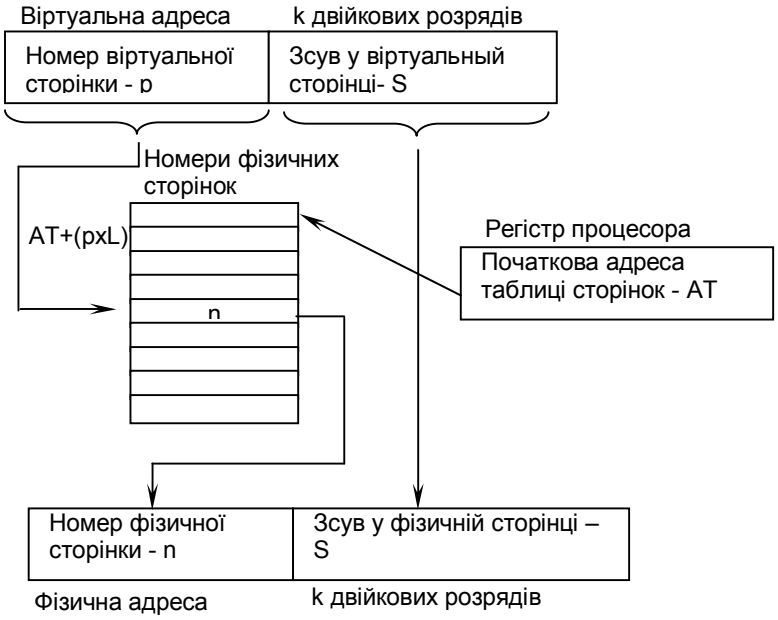


Рис. 6.7. Перетворення віртуальних сторінок на фізичні

Якщо відбувається звертання до пам'яті за деякою віртуальною адресою, апаратними схемами процесора виконуються наступні дії:

- Зі спеціального регістра процесора витягається адреса AT таблиці сторінок активного процесу. На підставі початкової адреси таблиці сторінок, номера віртуальної сторінки p (старші розряди віртуальної адреси) і довжини окремого запису в таблиці сторінок L (системна

константа) визначається адреса потрібного дескриптора в таблиці сторінок: a =AT+(px).

- Із цього дескриптора береться номер відповідної фізичної сторінки - n.
- До номера фізичної сторінки приєднується зсув s (молодші розряди віртуальної адреси).

Для зменшення часу перетворення адрес передбачений апаратний механізм одержання фізичної адреси з віртуальної.

Іншим важливим фактором, що впливає на продуктивність системи, є частота сторінкових переривань. На неї, у свою чергу, впливають розмір сторінки й прийняті в даній системі правила вибору сторінок для вивантаження й завантаження.

При виборі сторінки на вивантаження зміст критерію зводиться до одного: на диск виштовхується сторінка, до якої довше всього не буде звертань.

Однак, точно передбачити хід обчислювального процесу неможливо, а отже, і точно визначити сторінку, що підлягає вивантаженню. Рішення приймається на основі деяких емпіричних критеріїв.

Наприклад, якщо сторінка не використовувалася довгий час, робиться висновок про те, що вона не буде потрібна й зараз.

Найбільш популярним критерієм є число звертань до сторінки за останній період часу. ОС веде по кожній сторінці програмний лічильник. Коли виникає необхідність видалити яку-небудь сторінку з пам'яті, ОС знаходить сторінку, лічильник звертань якої має найменше значення. Всі лічильники періодично обнуляються.

Інтенсивність сторінкового обміну може бути знижена за рахунок так називаного **попереджуючого завантаження**, відповідно до якого, при виникненні сторінкового переривання у пам’ять завантажується не одна сторінка, а відразу декілька прилеглих до неї сторінок.

Концептуально передбачається, що якщо звертання відбулося за певною адресою, то велика ймовірність звертання за сусідніми адресами.

Найважливіше завдання – вибір оптимального розміру сторінки.

Якщо сторінка велика, то маємо великі втрати з фіктивних областей (у середньому половина останньої сторінки). Вибір розміру сторінки – складне оптимізаційне завдання. На практиці розмір сторінки вибирають у кілька кілобайт. Наприклад, у ОС на процесорах Pentium підтримують сторінки розміром 4096 байт (4 Кбайт).

Розмір сторінки також впливає на кількість записів у таблицях сторінок процесів. Так у сучасних процесорах максимальний обсяг віртуального адресного простору процесу не менше 4 Гбайт (2^{32}), тобто при розмірі сторінки 4 Кбайт (2^{12}) і довжині запису 4 байти для зберігання таблиці сторінок потрібно 4 Мбайт пам'яті.

Виходом з цієї ситуації є зберігання в пам'яті тільки тієї частини таблиці сторінок, що активно використовуються в даний період часу, тобто таблицю сторінок можна тимчасово витісняти з оперативної пам'яті.

Саме такий результат може бути досягнутий шляхом ускладнення структуризації віртуального адресного простору, при якому вся безліч віртуальних адрес процесу ділиться на **розділи**, а розділи діляться на **сторінки**. Всі сторінки мають однаковий розмір, а розділи містять однакову кількість сторінок.

Якщо розмір сторінки й кількість сторінок у розділі вибрати рівними степеням двійки (2^k й 2^n відповідно), то приналежність віртуальної адреси до розділу й сторінки, а також зсув усередині сторінки визначається дуже просто: молодші k двійкових розрядів дають *зсув*, наступні n розрядів – номер віртуальної сторінки, а старші розряди, що залишилися, містять номер розділу. Для кожного розділу будується власна таблиця сторінок. Кількість дескрипторів у таблиці і їхній розмір підбираються таким чином, щоб обсяг таблиці виявився рівним обсягу сторінки.

Наприклад, у процесорі Pentium при розмірі сторінки 4 Кбайт довжина дескриптора становить 4 байти, а кількість записів у таблиці сторінок, що

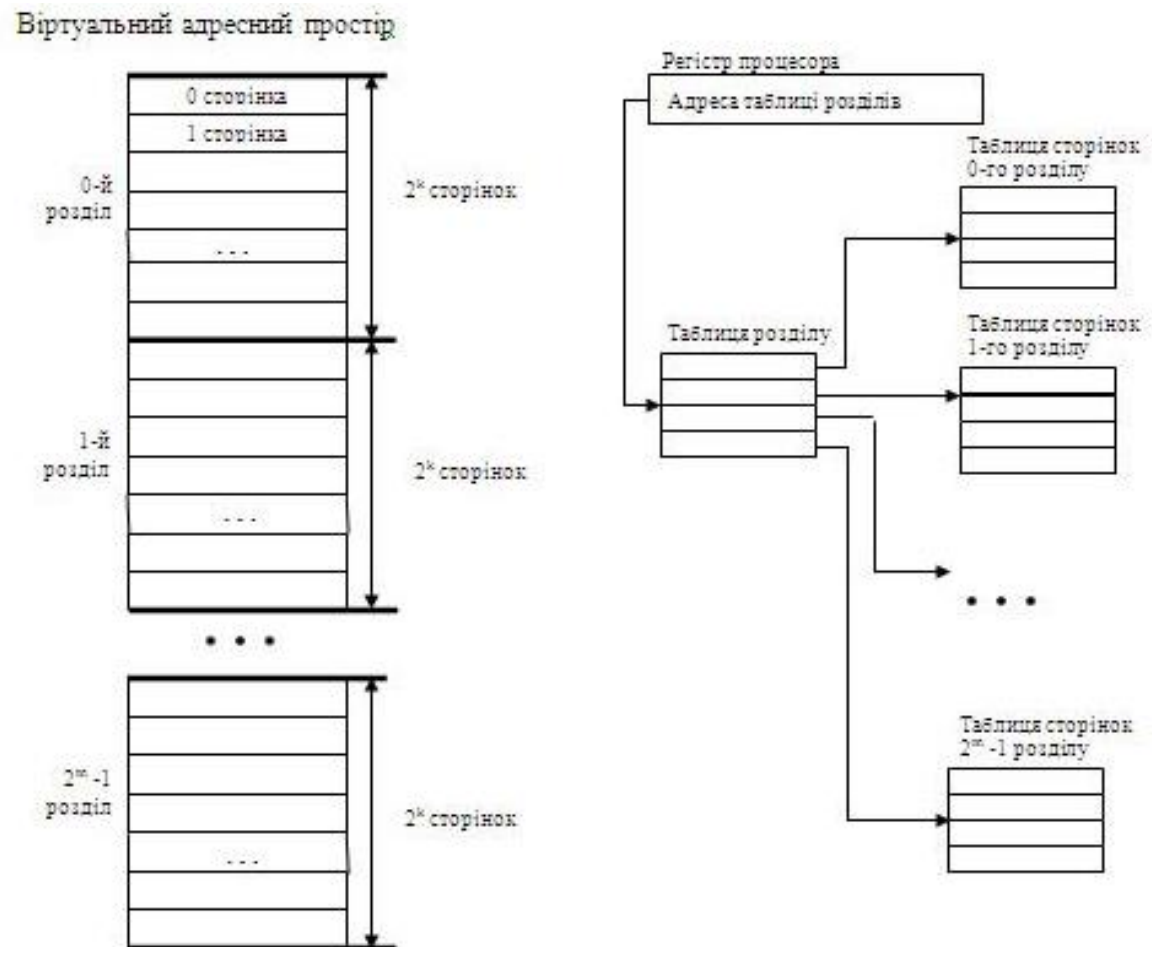


Рис. 6.8. Поділ віртуального простору на розділи і сторінки

вміщується на сторінку, дорівнює 1024. Кожна таблиця сторінок описується дескриптором, структура якого повністю збігається зі структурою дескриптора звичайної сторінки. Ці дескриптори зведені в **таблицю розділів**, називану також **каталогом сторінок**. Фізична адреса таблиці розділів активного процесу міститься в спеціальному регістрі процесора й тому завжди відома ОС. Сторінка, що містить таблицю розділів, ніколи не вивантажується з пам'яті. Вивантаження сторінок з таблицями сторінок дозволяє заощаджувати пам'ять, але при цьому призводить до додаткових витрат при одержанні фізичної адреси. Дійсно, може трапитися так, що та таблиця сторінок, що містить потрібний дескриптор, у цей момент вивантажена на диск. Тоді процес перетворення адреси припиняється до завантаження необхідної сторінки у пам'ять. Для зменшення ймовірності

відсутності сторінки в пам'яті використовують різні прийоми, основним з яких є кешування (розглядатиметься пізніше).

Простежимо більш докладно схему перетворення адрес для випадку дворівневої структуризації віртуального адресного простору (рис. 6.9):

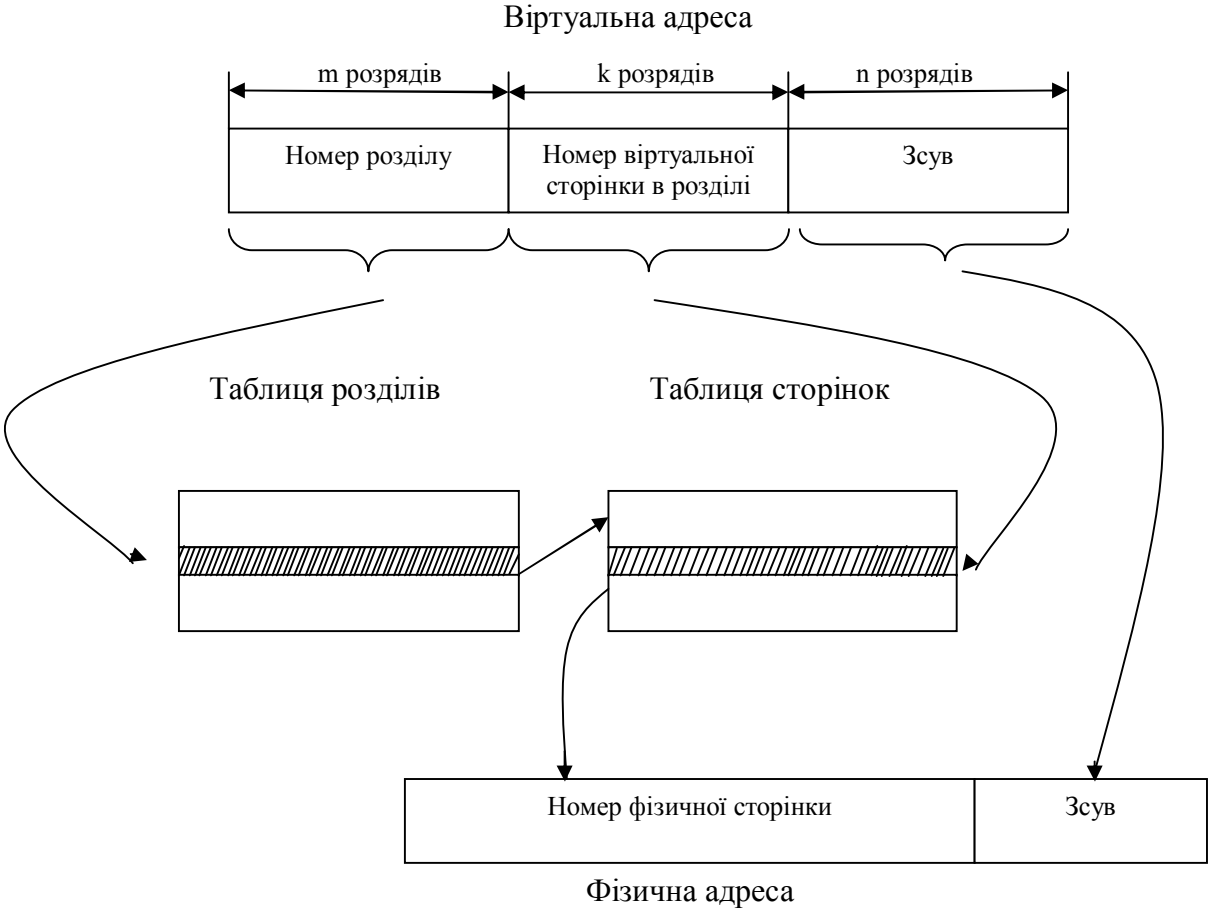


Рис. 6.9. Схема перетворення адрес для дворівневого адресного простору

1. Шляхом відкидання k+n молодших розрядів у віртуальній адресі визначається номер розділу, до якого належить дана віртуальна адреса.

2. За цим номером з таблиці розділів читається дескриптор відповідної таблиці сторінок. Перевіряється, чи перебуває дана таблиця сторінок у пам'яті. Якщо ні, відбувається сторінкове переривання й система завантажує потрібну сторінку з диску.

3. Далі із цієї таблиці сторінок читається дескриптор віртуальної сторінки, номер якої міститься в середніх **n** розрядах перетвореної віртуальної адреси. Знову виконується перевірка наявності даної сторінки в пам'яті й, при необхідності, її завантаження.

4. З дескриптора визначається номер (базова адреса) фізичної сторінки, у яку завантажена дана віртуальна сторінка. До номера фізичної сторінки приєднується зсув, узятий з k молодших розрядів віртуальної адреси. Як результат, отримаємо шукану фізичну адресу.

Зауважимо, що сторінковий розподіл пам'яті може бути реалізований в спрощеному варіанті, без вивантаження сторінок на диск. У цьому випадку всі віртуальні сторінки всіх процесів постійно перебувають в оперативній пам'яті. Такий варіант не має переваг роботи з віртуальною пам'яттю великого обсягу, але дозволяє успішно боротися із фрагментацією фізичної пам'яті. Ніколи не утворяться невикористовувані залишки. Такий режим використовується в спеціалізованих ОС, коли потрібна висока реактивність системи.

Сегментний розподіл

При сторінковій організації віртуальний адресний простір процесу поділяють на рівні частини *механічно,* без урахування змісту даних.

Однак, розбиття віртуального адресного простору на «осмислені» частини робить принципово можливим спільне використання фрагментів програм різними процесами. Наприклад, двом процесам потрібна одна й та сама підпрограма, що до того ж має властивість реєнтерабельності. Тоді коди цієї підпрограми можуть бути оформлені у вигляді окремого сегмента й включені у віртуальні адресні простори обох процесів.

При відображенні у фізичну пам'ять сегменти, що містять коди підпрограми з обох віртуальних просторів, проектуються на ту саму область фізичної пам'яті. Таким чином, обидва процеси одержать доступ до однієї й тієї ж копії підпрограми.

Віртуальний адресний простір процесу поділяють на частині — **сегменти**.

Розмір сегментів визначається з урахуванням змістовності інформації, що знаходиться в них. Окремий сегмент може являти собою підпрограму, масив

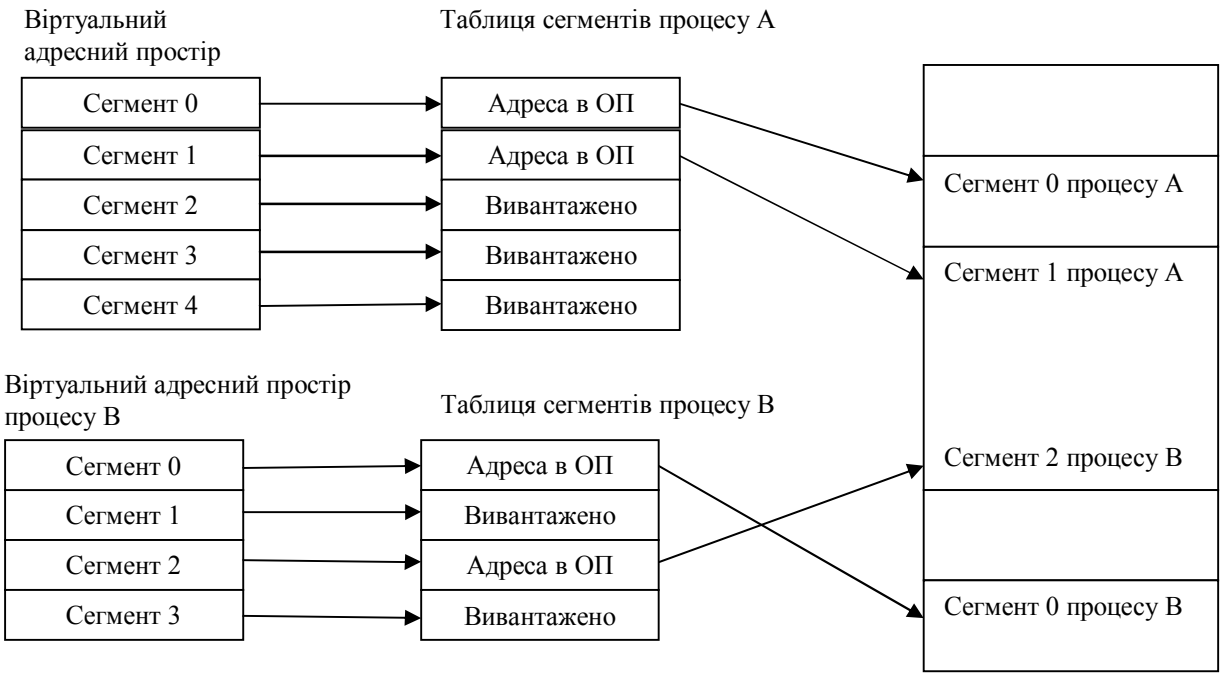


Рис. 6.10. Сегментний розподіл

даних і т.д. Поділ віртуального адресного простору на сегменти здійснюється компілятором на основі вказівок програміста або за замовчуванням, на основі прийнятих у системі угод. Максимальний розмір сегмента визначається розрядністю віртуальної адреси, наприклад, при 32-розрядній організації процесора він дорівнює 4 Гбайт. При цьому максимально можливий віртуальний адресний простір процесу являє собою набір з N віртуальних сегментів, кожен розміром по 4 Гбайт. У кожному сегменті віртуальні адреси перебувають у діапазоні від 00000000_{16} до $FFFFFFFF_{16}$.

Сегменти не впорядковуються відносно один одного, так що загальної для сегментів лінійної віртуальної адреси не існує, і віртуальна адреса задається парою чисел: номером сегмента й лінійною віртуальною адресою.

При завантаженні процесу в оперативну пам'ять там міститься тільки частина його сегментів, а повна копія віртуального адресного простору перебуває в дисковій пам'яті. Для кожного сегмента, що завантажується, ОС підшукує неперервну ділянку вільної пам'яті достатнього розміру. Суміжні у віртуальній пам'яті сегменти одного процесу можуть займати в оперативній

пам'яті несуміжні ділянки (це очевидно). Якщо під час виконання процесу відбувається звертання по віртуальній адресі (що відноситься до сегмента), що у цей момент відсутня у пам'яті, то відбувається переривання. ОС припиняє активний процес, запускає на виконання наступний процес із черги, а паралельно організує завантаження потрібного сегмента. При відсутності в пам'яті місця, ОС вибирає сегмент на вивантаження.

На етапі створення процесу під час завантаження його образу в оперативну пам'ять система створює **таблицю сегментів** процесу (подібну таблиці сторінок), у якій для кожного сегмента вказується:

- базова фізична адреса сегмента в оперативній пам'яті;
- розмір сегмента;
- правила доступу до сегмента;
- ознаки модифікації, присутності й звертання до даного сегмента, а також деяка інша інформація.

Якщо віртуальні адресні простори декількох процесів включають той самий сегмент, то в таблицях сегментів цих процесів вказують посилання на ту ділянку оперативної пам'яті, у яку даний сегмент завантажується в єдиному екземплярі.

Механізми перетворення адрес у сторінковому й сегментному способах організації пам'яті досить схожі. Однак у них є й істотні відмінності. Ці відмінності є наслідком того, що сегменти, на відміну від сторінок, мають довільний розмір.

Віртуальну адресу при сегментній організації можна представити парою (**g**, **s**), де **g** – номер сегмента, а **s**- зсув.

Сторінки мають однаковий розмір, а їхні початкові адреси кратні розміру сторінок. Тому ОС заносить у таблиці сторінок не повні адреси, а номери фізичних сторінок, які збігаються зі старшими розрядами базових адрес.

У загальному випадку, сегмент може розташовуватися у фізичній пам'яті, починаючи з будь-якої адреси. Отже, для визначення місця розташування в пам'яті необхідно задавати його повну початкову фізичну адресу (рис. 6.11).

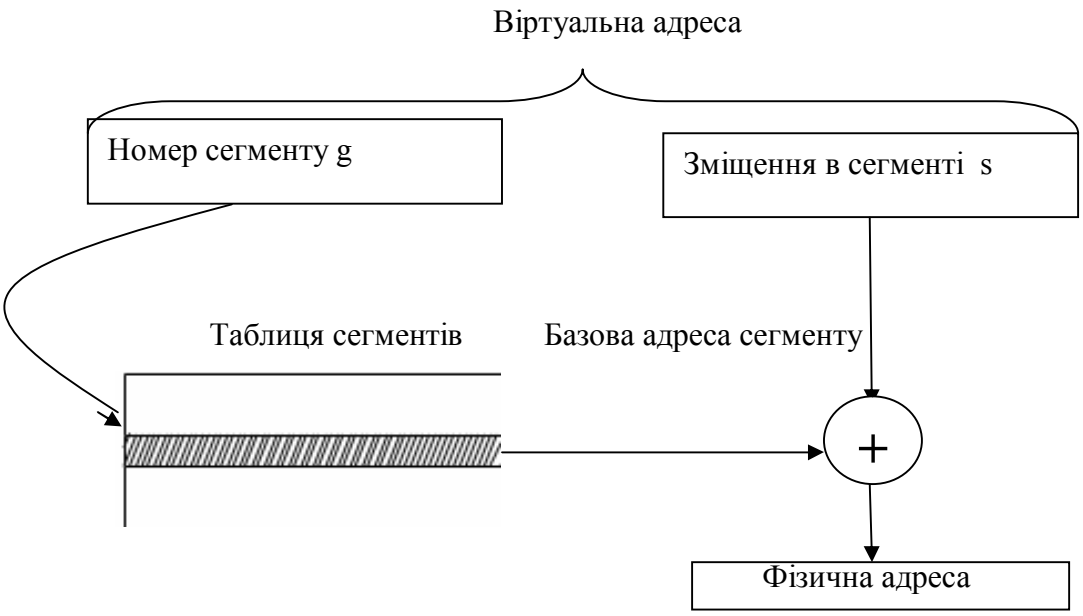


Рис.6.11. Перетворення віртуальної адреси на фізичну

Використання операції додавання сповільнює процедуру перетворення віртуальної адреси у фізичну, порівняно зі сторінковою організацією (там просте об'єднання старших і молодших розрядів).

Іншим недоліком сегментного розподілу є надмірність. При сегментній організації одиницею переміщення між пам'яттю й диском є сегмент, об’єм якого, як правило, більший, ніж сторінка. Але у багатьох випадках для роботи програми не потрібно завантажувати повністю увесь сегмент, достатньо було б однієї або двох сторінок. Аналогічно, при відсутності вільного місця не слід вивантажувати цілий сегмент, якщо можна обмежиться декількома сторінками. Але головний недолік – це фрагментація, що виникає через непередбачуваність розмірів сегментів.

Найбільш істотною відмінністю сегментної організації від сторінкової є можливість визначення диференційованих прав доступу процесу до його сегментів. Наприклад, сегмент даних вихідної інформації може мати права доступу «тільки читання», а сегмент результатів – «читання й запис». Ця властивість визначає перевагу перед сторінковою організацією.

Сегментно-сторінковий розподіл

Даний метод є спробою об'єднати достоїнства обох методів, а саме сегментної та сторінкової організації.

Тут, як і при сегментній організації пам'яті, віртуальний адресний простір процесу розділений на сегменти. Це дозволяє визначати різні права доступу до різних даних. Переміщення даних між пам'яттю й диском здійснюється не сегментами, а сторінками. Для цього кожен сегмент і фізична пам'ять діляться на сторінки рівного розміру, що дозволяє мінімізувати фрагментацію.

У більшості сучасних реалізацій сегментно-сторінкової організації діапазонів адрес всі віртуальні сегменти утворюють один неперервний лінійний віртуальний адресний простір (рис. 6.12).

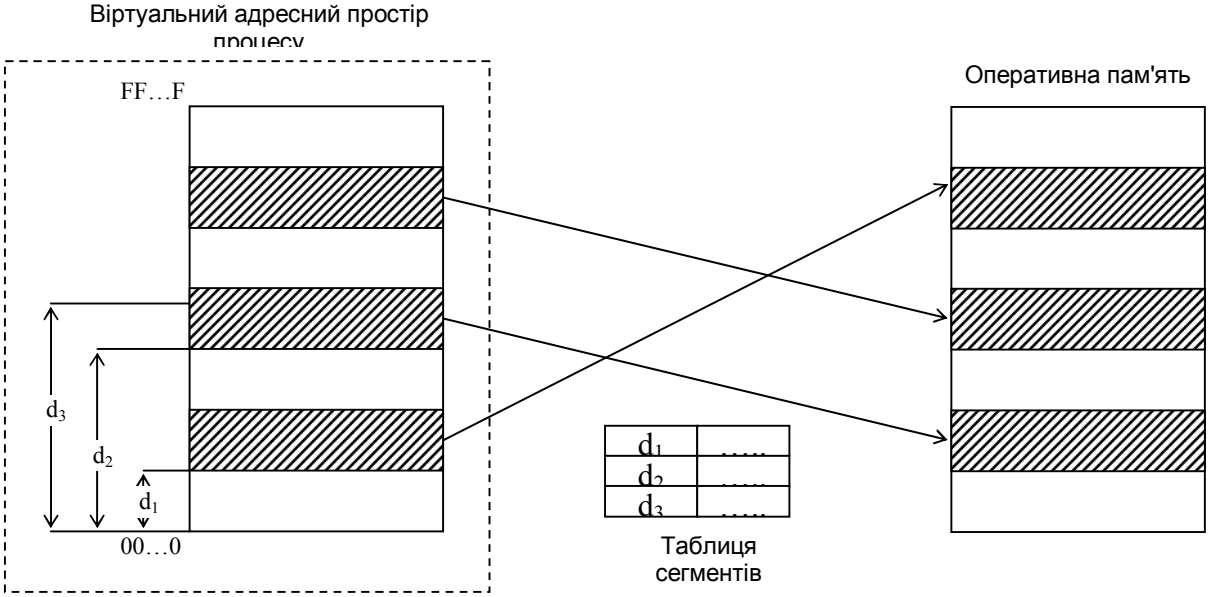


Рис. 6.12. Сегментно – сторінковий розподіл

Координати байта при цьому можна задати двома способами. По-перше, лінійною віртуальною адресою, що дорівнює зсуву даного байта щодо границі загального лінійного віртуального простору, по-друге, парою чисел, одне з яких є номером сегмента, а інше – зсувом щодо сегмента.

При цьому, на відміну від сегментної моделі, для однозначного визначення віртуальної адреси другим способом, необхідно вказати також початкову віртуальну адресу сегмента з даним номером.

Найчастіше використовується другий спосіб, тому що він дозволяє безпосередньо визначити приналежність адреси деякому сегменту й перевірити права доступу процесу до нього.

Для кожного процесу ОС створює окрему таблицю сегментів, у якій утримуються дескриптори (описувачі) всіх сегментів процесу. Опис сегмента включає призначені йому права доступу й інші характеристики, подібні тим, які містяться в дескрипторах сегментів при сегментній організації пам'яті.

Однак є і відмінності. У полі базової адреси вказується не початкова фізична адреса сегмента, а початкова лінійна віртуальна адреса сегмента в просторі віртуальних адрес. Його наявність дозволяє однозначно перетворити адресу, задану парою (g, s) у лінійну віртуальну адресу байта, що потім перетвориться у фізичну адресу сторінковим механізмом.

Розподіл загального лінійного віртуального адресного простору процесу й фізичної пам'яті на сторінки здійснюється так само, як це робиться при сторінковій організації пам'яті.

Базові адреси таблиці сегментів і таблиці сторінок процесу є частиною його контексту. При активізації процесу ці адреси завантажуються в спеціальні регістри процесів і використовуються механізми перетворення адрес.

Перетворення віртуальної адреси у фізичну відбувається у два етапи (рис. 6.13).

1. На першому етапі працює механізм сегментації. Вихідна віртуальна адреса пари (g, s) перетворюється в лінійну віртуальну адресу. Для цього на підставі базової адреси сегментів і номера сегмента обчислюється поле дескриптора й виконується перевірка можливості виконання заданої операції.

Якщо доступ до сегмента дозволений, то обчислюється лінійна віртуальна адреса шляхом додавання базової адреси сегмента, прочитаного з дескриптора, і зсуву, заданого у вихідній віртуальній адресі.

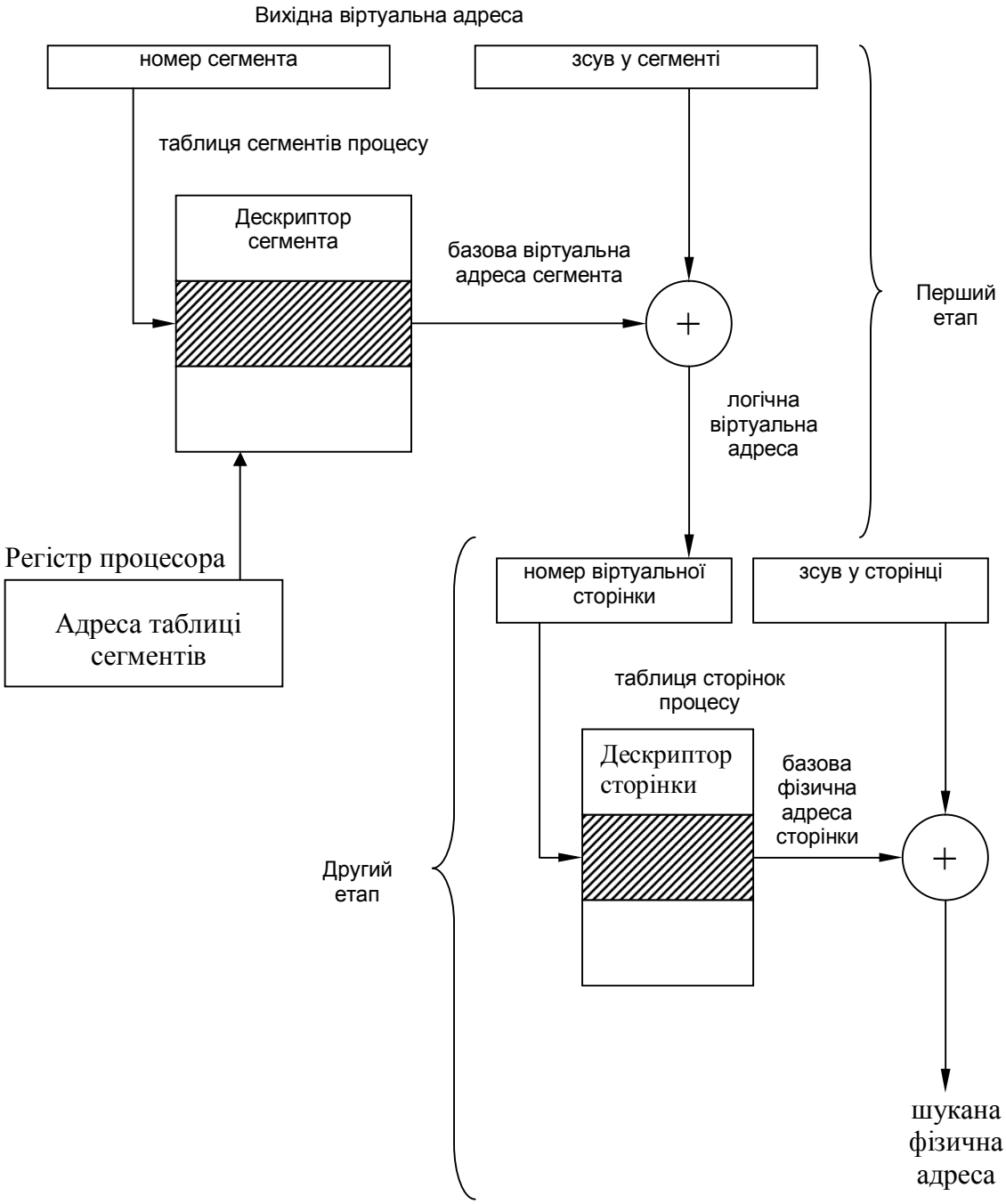


Рис. 6.13. Перетворення віртуальної адреси у фізичну

2. Другий етап використовує сторінковий механізм. Лінійна віртуальна адреса перетворюється у фізичну. Як результат перетворення, лінійна віртуальна адреса представляється у тому вигляді, у якому вона

використовувалась при сторінковій організації пам'яті (пара: номер сторінки, зсув у сторінці). Далі, як вже визначалося раніше для сторінкової організації.

Старші розряди лінійної віртуальної адреси, що містять номер віртуальної сторінки, замінюються номером фізичної сторінки, узятим з таблиці сторінок, а молодші розряди віртуальної адреси, що містять зсув, залишаються без зміни.

Як слідує з розгляду, механізми сегментації і сторінковий діють досить незалежно друг від друга.

Тому не важко уявити собі реалізацію пам'яті, коли механізм сегментації виконується, як вищезазначено, а сторінковий механізм реалізується за дворівневою схемою: віртуальний адресний простір ділиться спочатку на розділи, а вже потім на сторінки.

У такому випадку, перетворення віртуальної адреси у фізичну відбувається в кілька етапів.

Спочатку механізм сегментації звичайним чином, використовуючи таблицю сегментів, обчислює лінійну віртуальну адресу. Потім із цієї адреси обчислюється номер розділу, номер сторінки й зсув. За номером розділу з таблиці розділів визначається адреса таблиці сторінок, а потім за номером віртуальної сторінки з таблиці сторінок визначається номер фізичної сторінки, до якої приєднується зсув.

Саме цей підхід реалізований у процесорах i386, i486 й Pentium.

Розглянемо ще одну схему керування пам'яттю, засновану на сегментно-сторінковому підході.

Відмінність цього підходу полягає в тому, що віртуальні сторінки нумеруються не в межах усього адресного простору процесу, а в межах сегмента. Віртуальна адреса, у цьому випадку, визначається трійкою (номер сегмента, номер сторінки, зсув у сторінці).

Завантаження процесу виконується посторінково. Для кожного процесу створюється своя таблиця сегментів, а для кожного сегмента – своя таблиця сторінок. Адреса таблиці сегментів завантажується в спеціальний регістр

процесора. Таблиці сторінок повністю аналогічні таблицям сторінок у попередньому випадку.

Таблиця сегментів має істотні відмінності. Вона складається з дескрипторів сегментів, які, замість інформації про розташування сегментів у віртуальному адресному просторі, містять опис розташування таблиць сторінок у фізичній пам'яті (рис. 6.14).

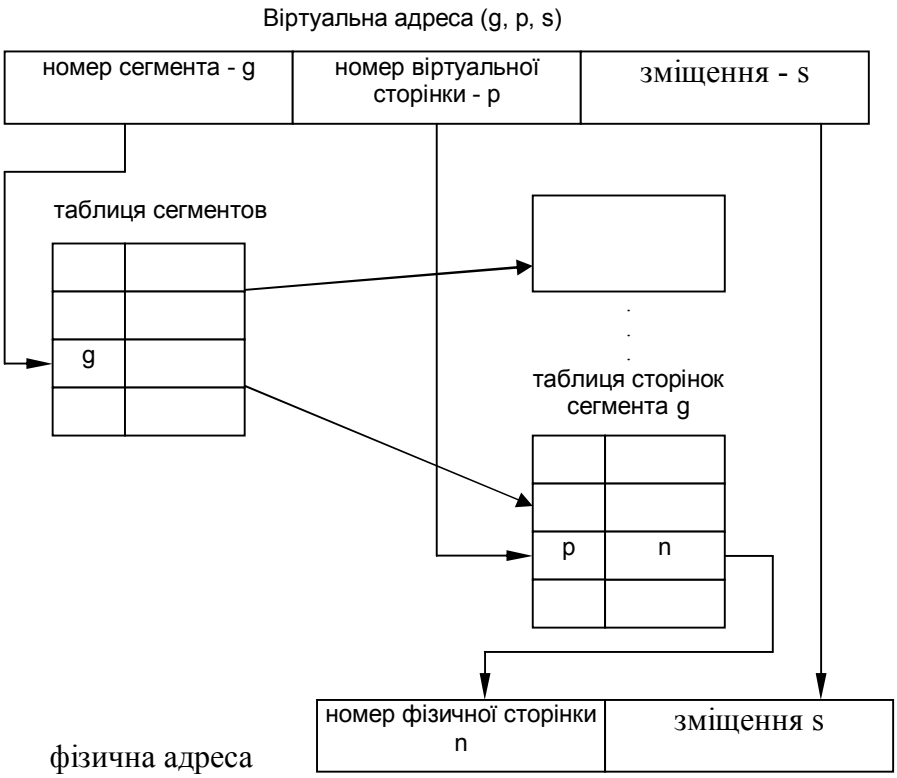


Рис. 6.14. Схема для другого випадку

1. За номером сегмента, заданому у віртуальній адресі, з таблиці сегментів визначається фізична адреса відповідної таблиці сторінок.

2. За номером віртуальної сторінки, заданої у віртуальній адресі, з таблиці сторінок читається дескриптор, у якому зазначений номер фізичної сторінки.

3. До номера фізичної сторінки приєднується молодша частина віртуальної адреси – зсув.

Поділювані сегменти пам'яті

Прикладом застосування поділюваної області пам'яті може бути використання її у якості буфера при міжпроцесному обміні даними. Один процес пише в поділювану область, інший – читає.

Для організації поділюваного сегмента при наявності системи віртуальної пам'яті досить помістити його у віртуальний адресний простір кожного процесу, якому потрібний доступ до даного сегмента, а потім настроїти параметри відображення цих віртуальних сегментів так, щоб вони відповідали однієї й тій самій області оперативної пам'яті. Деталі такого настроювання залежать від пам'яті.

«Влучення» віртуальних сегментів на загальну частину оперативної пам'яті досягається або за рахунок погодженого настроювання ОС численних дескрипторів для процесів, або переміщенням єдиного поділюваного віртуального сегмента у загальну частину віртуального адресного простору процесів, яку звичайно використовують для модулів ОС. У цьому випадку, настроювання дескриптора сегмента й дескрипторів сторінок виконується одноразово, а всі процеси користуються такою побудовою й спільно використовують частину оперативної пам'яті. При роботі з поділюваними сегментами пам'яті необхідно дотримуватися загальних правил використання поділюваних ресурсів: семафорів, моніторів тощо.

Для відмінності поділюваних сегментів пам'яті від індивідуальних сегментів дескриптор сегмента повинен містити поле, що має два значення: shared (поділюваний) або private (індивідуальний).

Поділювані сегменти вивантажуються на диск системою віртуальної пам'яті за тим самим алгоритмом і за допомогою тих самих механізмів, що й індивідуальні сегменти.

6.4 Контрольні запитання до розділу 6

1. Які функції по керуванню пам’яттю виконує ОС?

2. Які типи адрес існують на різних етапах життєвого циклу програми?

3. Які підходи використовують при віртуалізації пам’яті?

4. Якими класами представлені процеси віртуалізації памяті?

5. Які існують методи розподілу пам’яті?

6.Які особливості алгоритму розподілу пам’яті динамічними розділами?

7. Які особливості розподілу пам’яті переміщувальними розділами?

8. Яка різниця між спулінгом і віртуалізацією пам’яті?

9. Який алгоритм перетворення віртуальної адреси у фізичну використовують при сторінковій організації пам’яті?

10. Який алгоритм перетворення віртуальної адреси у фізичну використовують при сегментній організації пам’яті?

11. Який алгоритм перетворення віртуальної адреси у фізичну використовують при сегментно-сторінковій організації пам’яті?

12. Як організувати поділювальні сегменти пам’яті?

Розділ 7. Кешування даних

7.1 Визначення кешування пам’яті

Пам'ять ЕОМ – це ієрархія запам’ятовуючих пристроїв (ЗП), що відрізняються середнім часом доступу до даних, обсягом і вартістю зберігання 1 біта інформації.

Фундамент цієї піраміди – пам'ять на жорстких дисках, але час доступу до диску обчислюється мілісекундами.

Оперативна пам'ять має час доступу 10-20 наносекунд (від декількох мегабайт до декількох гігабайт)

Надоперативна пам'ять (десятки - сотні кілобайт) – 8 нсек.

Внутрішні регістри процесора – кілька десятків байт з часом доступу 2-3 наносекунди.

Кеш пам'ять або просто **кеш –** спосіб спільного функціонування двох типів запам'ятовуючих пристроїв, що відрізняються часом доступу й вартістю зберігання даних, який, за рахунок динамічного копіювання у швидкі ЗП найчастіше використовуваної інформації з «повільного» ЗП, дозволяє, з одного боку, зменшити середній час доступу до даних, з іншого боку – заощадити дорожчу швидкодіючу пам'ять.

Особливістю кешування є те, що система не вимагає ніякої зовнішньої інформації про інтенсивність використання даних. Ані користувачі, ані програми не приймають ніякої участі в переміщенні даних із ЗП одного типу в ЗП іншого типу, все це робиться автоматично системними засобами.

Кешем часто називають не тільки спосіб організації двох типів запам'ятовуючих пристроїв, але й один із пристроїв – «швидкий» ЗП. Він дорожчий й порівняно невеликого обсягу на противагу «повільному» ЗП – оперативній пам'яті.

Якщо **кешування** використовують для зменшення середнього часу **доступу до оперативної пам'яті**, то в якості КЕШа використовують більш дорогу й швидкодіючу статичну пам'ять.

Якщо **кешування** використовується системою введення-виведення для прискорення доступу до даних, що **зберігаються на диску,** роль кеш-пам'яті виконують буфери, реалізовані в оперативній пам'яті.

Віртуальну пам'ять теж можна розглядати як окремий випадок **кешування**.

7.2 Принцип дії кеш пам'яті

Вміст кеш пам'яті являє собою сукупність записів про всі завантажені в неї елементи даних з основної пам'яті. Кожен запис включає:

- елементи даних;
- адресу елемента має в основній пам'яті;
- додаткову інформацію, що використовується для реалізації алгоритму та містить ознаку модифікації й ознаку дійсності даних.

При кожному звертанні до основної пам'яті за фізичною адресою проглядається вміст кеш пам'яті з метою визначення, чи не перебувають там потрібні дані.

Кеш-пам'ять не є адресуємою, тому пошук потрібних даних здійснюється за вмістом – за взятим із запиту значенням поля адреси в оперативній пам'яті.

Далі можливі два варіанти розвитку (рис. 7.1):

- дані виявлені в кеш пам'яті, тобто зроблено кеш-влучання (cache-hit), вони зчитуються й передаються джерелу запиту;
- потрібні дані відсутні, тобто відбувся кеш-промах (cache-miss), вони зчитуються з основної пам'яті, передаються джерелу запиту й одночасно копіюються в кеш-пам'ять.

Ефективність кешування залежить від ймовірності влучення в кеш.

Якщо позначити ймовірність кеш-влучання через p, а час доступу до основної пам'яті через t1, час доступу до кеш через t2, то за формулою повної ймовірності середній час доступу буде дорівнювати:

$$t=t2p+t1(1-p)$$

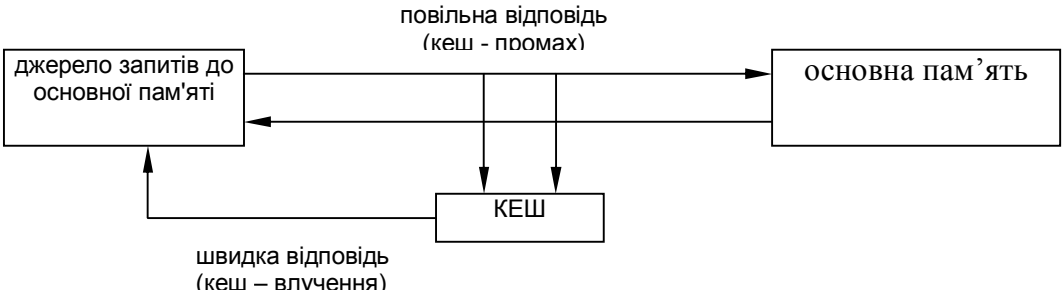


Рис. 7.1. Принцип кешування

Якщо p=1, час доступу дорівнює t_2.

Ймовірність виявлення даних у кеші залежить від різних факторів, таких як:

- обсяг кешу;
- обсяг кешуємої пам'яті;
- алгоритму заміщення даних у кеші;
- особливостей виконуваної програми й т.п.

На практиці відсоток влучень виявляється досить високим – порядку 90%. Такий відсоток обумовлюється наявністю в даних об'єктивних властивостей, таких як просторової й тимчасової локальності.

Просторова локальність. Якщо відбулося звертання за деякою адресою, то з високою ймовірністю найближчим часом відбудеться звертання за сусідніми адресами.

Часова локальність. Якщо відбулося звертання за деякою адресою, то наступне звертання за тією ж адресою з великою ймовірністю відбудеться найближчим часом.

Оскільки при виконанні програми дуже висока ймовірність, що команди вибираються з пам'яті одна за одною із сусідніх комірок, має сенс

завантажувати в кеш цілий фрагмент програми. Аналогічно й з масивами даних.

У процесі роботи вміст кеш-пам'яті постійно оновлюється. Витіснення даних означає або просто оголошення вільною деякої області кеш-пам'яті (скидання біта дійсності), якщо дані не змінювалися, або, на додаток до цього, копіювання даних в основну пам'ять, якщо вони були модифіковані.

Наявність у комп'ютері двох копій даних: в основній пам'яті й у кеші – породжує проблему узгодження даних.

Існують два підходи до вирішення цієї проблеми:

- наскрізний запис (write through). При запиті до основної пам'яті (у тому числі при записі) переглядають кеш. Якщо дані за запитуваною адресою відсутні, запис виконується тільки в основну пам'ять. Якщо дані перебувають у кеші, запис робиться у кеш й у пам'ять.
- зворотний запис (write back). Виконується перегляд кешу, якщо даних там немає, то запис робиться в основну пам'ять. У противному випадку, запис робиться тільки в кеш. При цьому встановлюється ознака модифікації. При витісненні даних з кешу вони будуть переписані в основну пам'ять.

7.3 Способи відображення основної пам'яті на кеш

Алгоритми пошуку й заміщення даних у кеші залежать від того, як основна пам'ять відображається на кеш.

Використовують дві схеми відображення:

- випадкове відображення;
- детерміноване відображення.

При випадковому відображенні елемент оперативної пам'яті може бути розміщений у довільному місці кеш пам'яті. Він роміщується там разом зі своєю адресою в оперативній пам'яті. Пошук інформації здійснюється за цією адресою. Процедури простого перебору адрес потребують великих часових витрат.

Тому використовується асоціативний пошук, при якому порівняння виконується не послідовно з кожним записом у кеші, а паралельно з усіма його записами. Ознака, за якою виконується порівняння, називають **тегом** (tag). У даному випадку, тегом є адреса даних в оперативній пам'яті.

Електронна реалізація такого пошуку значно здорожчує кеш-пам'ять. Тому цей метод використовується для забезпечення високого відсотка влучення при невеликому обсязі кеш пам'яті.

У кешах на основі випадкового відображення витіснення старих даних відбувається тільки в тому випадку, коли вся кеш пам'ять заповнена і немає вільного місця. Вибір даних на вивантаження ґрунтується на тих самих принципах, що й при заміщенні сторінок (давно немає звертань, найменше звертань і т.д.). При **детермінованому** способі відображення будь-який елемент основної пам'яті відображається в одне й те ж саме місце кеш пам'яті. У цьому випадку, кеш пам'ять розділена на рядки, кожен з яких призначений для зберігання одного запису про один елемент даних і має свій номер.

Між номерами рядків кеш пам'яті й адресами оперативної пам'яті встановлюється відповідність «**один до багатьох**»: одному номеру рядка відповідає декілька (досить багато) адрес оперативної пам'яті.

Як функцію відображення можна використовувати просте виділення декількох розрядів з адреси оперативної пам'яті, які інтерпретуються як номер рядка кеш-пам'яті. Таке відображення називають **прямим** (рис. 7.2). Наприклад, кеш розраховано на 1024 записи (1024 рядків). Тоді будь-яка адреса оперативної пам'яті може бути відображена на адресу кеш пам'яті простим відділенням 10 двійкових розрядів.

При пошуку даних у кеші використовується швидкий прямий доступ до запису за номером рядка, отриманим з адреси оперативної пам'яті із запиту. Крім того, виконується додаткова перевірка на збіг тегу з відповідною частиною адреси із запиту.

При збігу тегу з відповідною частиною адреси із запиту констатується кеш-влучення. Якщо немає збігу, констатується кеш-промах, і дані зчитуються з ОП і копіюються в кеш.

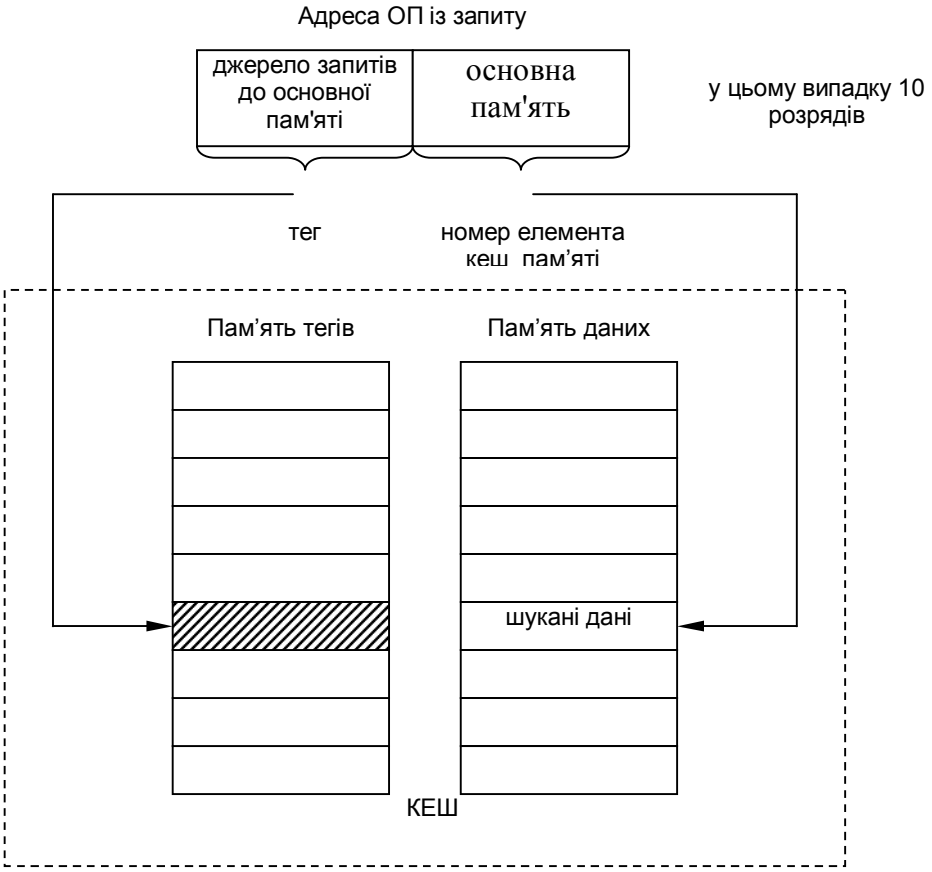


Рис. 7.2. Пряме відображення пам'яті на кеш

У багатьох сучасних процесорах кеш пам'ять будується на основі поєднання цих двох підходів, що дозволяє знайти певний компроміс у швидкодії.

При **змішаному підході** (рис. 7.3) довільна адреса оперативної пам'яті відображається не на одну адресу кеш пам'яті (як це характерно для прямого відображення) і не на довільну адресу кеш пам'яті (як це робиться при випадковому відображенні), а на певну групу адрес. Всі групи пронумеровані. Пошук у кеші здійснюється спочатку за номером групи, отриманим з адреси ОП із запиту, а потім у межах групи шляхом асоціативного перегляду всіх записів групи на предмет збігу старших частин адрес ОП.

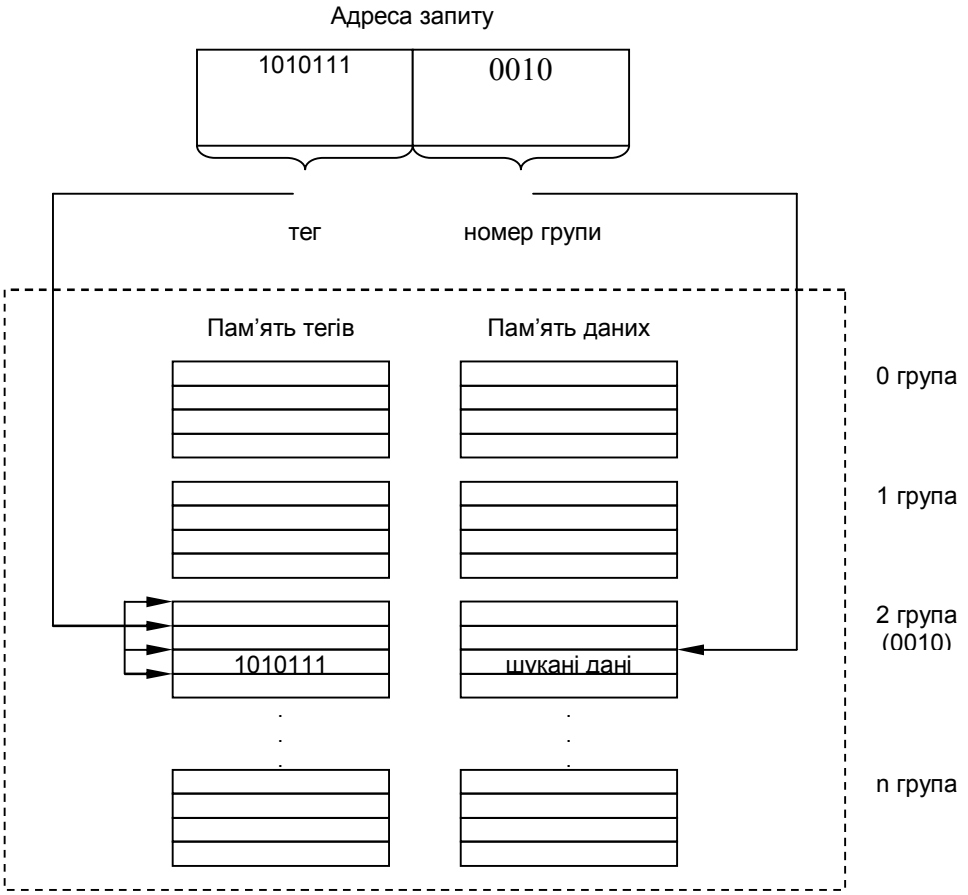


Рис. 7.3. Комбінування прямого й
випадкового відображення пам’яті у кеш

При промаху дані копіюються за будь-якою вільною адресою з визначеної групи.

Якщо вільних адрес у групі немає, то виконується витіснення даних. Оскільки кандидатів на вивантаження кілька – всі записи з даної групи – алгоритм заміщення може врахувати інтенсивність звертання до даних і, тим самим, підвищити ймовірність влучень у майбутньому.

Як бачимо, кеш проглядається тільки з метою узгодження вмісту кеша й основної пам'яті. Якщо відбувається промах, то запити на запис не викликають ніяких змін кеша.

У деяких реалізаціях кеш пам'яті, при відсутності даних у кеші, вони копіюються туди з основної пам'яті незалежно від того, виконується запит на читання чи запис.

Відповідно до описаної логіки роботи кеш пам'яті, при виникненні запиту спочатку проглядається кеш, а потім, якщо відбувся промах, виконується звертання до основної пам'яті.

Однак, часто реалізується й інша схема роботи кеша: **пошук** у кеші й основній пам'яті **починається одночасно,** потім у результаті перегляду кеша, операція в основній пам'яті або триває, або переривається.

У ряді обчислювальних систем використовується дворівневе кешування (рис. 7.4).

Кеш першого рівня має менший обсяг і більш високу швидкодію, ніж кеш другого рівня. Кеш другого рівня відіграє роль основної пам'яті стосовно кеша першого рівня.

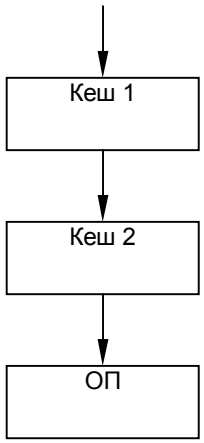


Рис. 7.4. Дворівневе кешування

Спочатку робиться спроба виявити дані в кеші першого рівня. Якщо відбувся промах, пошук триває в кеші другого рівня. Якщо потрібні дані відсутні й тут, тоді відбувається зчитування даних з основної пам'яті.

При зчитуванні даних з ОП відбувається їхнє копіювання в кеш другого рівня, а якщо дані зчитуються з кеша другого рівня, то вони копіюються в кеш першого рівня.

Кеши різних рівнів можуть погоджувати дані різними способами. Наприклад, у процесорі Pentium кеш першого рівня використовує кеш першого рівня – зворотний запис.

Зауважимо, що розглядали системи, у яких на кожному рівні є тільки один кеш. Однак існує цілий ряд систем – розподілених систем обробки інформації, у яких на кожному рівні є кілька кешів.

7.4 Контрольні запитання до розділу 7

1.Наведіть визначення КЕШ-пам’яті.

2. Який принцип дії КЕШ – пам’яті?

3. Яким чином вирішується проблема узгодження даних між кешем та основною пам’ятью?

4. Які використовуються схеми відображення основної пам’яті на кеш?

5. Який існує алгоритм комбінування прямого і випадкового відображення основної пам’яті на кеш?

6. Яким чином організовано багаторівневе кешування?

Розділ 8. Введення-виведення і файлова система

Основними компонентами підсистеми введення-виведення є драйвери, що керують зовнішніми пристроями, і файлові системи.

До **підсистеми введення-виведення** можна також віднести й **диспетчер переривань**; проте це досить умовно, оскільки він обслуговує не тільки модулі підсистеми введення-виведення, але також і планувальник або диспетчер потоків.

Файлову систему доцільно розглядати разом з іншими компонентами вводу - виводу по двох причинах:

- файлова система активно використовує введення-виведення;
- модель файлу може бути в основі більшості механізмів доступу до пристроїв вводу - виводу.

8.1 Завдання ОС по керуванню файлами й пристроями

Підсистема введення-виведення мультипрограмної ОС вирішує наступні основні завдання:

- організація паралельної роботи пристроїв введення-виведення й процесора;
- узгодження швидкостей обміну й кешування даних;
- поділ пристроїв і даних між процесами;
- забезпечення зручного логічного інтерфейсу між пристроями й іншою частиною системи;
- підтримка широкого спектра драйверів з можливістю простого включення в систему нового драйвера;
- динамічне завантаження й вивантаження драйверів;
- підтримка декількох файлових систем;
- підтримка синхронних й асинхронних операцій введення-виведення.

8.2 Основні поняття та концепція організації введення-виведення

Складність реалізації введення-виведення полягає у тому, що перед проектувальниками постає задача підтримки широкого різноманіття пристроїв, з одного боку, а, з другого боку, – сприяння створенню ефективного віртуального інтерфейсу незалежно від специфіки пристроїв введення-виведення та принципів їх розподілу між виконуваними задачами.

Тобто, система введення-виведення повинна бути універсальною, щоб об’єднувати різні пристрої від миші до графічних дисплеїв та накопичувачів на магнітних дисках. З іншого боку, доступ до цих пристроїв повинен бути таким, щоб паралельно виконувані задачі не заважали одна одній, якщо вони використовують одні і ті ж самі пристрої.

Тому завжди використовують наступний найголовніший принцип: операції введення-виведення об’являють привілейованими і виконують тільки кодами даної ОС.

Для забезпечення цього принципу вводять режим супервізора та користувача. Як правило, виконання введення-виведення – це розподілювальні ресурси. Тому, у загальному випадку, їх використання потребує синхронізації, хоча можуть існувати і нероздільні пристрої, наприклад, принтер.

Можна назвати три основні принципи, за якими не можна кожній окремій користувацькій програмі звертатись до зовнішнього пристрою безпосередньо:

1) Необхідність вирішення можливих конфліктів доступу до пристроїв. Наприклад, якщо дві або декілька програм будуть друкувати свої результати, то, окрім проблеми звичайної синхронізації доступу, виникає додаткова проблема: як розділити між собою ті результати, які будуть друкуватися упереміш.

2) Бажання збільшити ефективність використання цих ресурсів. Наприклад, у накопичувача на магнітних дисках час допуску до

необхідної доріжки та звертання до відповідного сектору може на декілька порядків перевищувати час пересилання даних.

Тому, якщо задачі звертаються почергово до циліндрів, які далеко розташовані один від одного, ефективний час роботи накопичувача буде суттєво зменшений.

3) Похибки у програмах введення-виведення можуть призвести до зупинки усіх процесів, бо частина операції введення-виведення виконується для самої ОС. Власне у ряді ОС операції введення-виведення для потреб ОС мають суттєві привілеї перед такими ж операціями для потреб звичайних процесів. Системний код, що керує операціями введення-виведення, дуже щільно відпрацьовується для підвищення надійності та ефективного використання обладнання.

Тобто управління введенням-виведенням виконується операційною системою, власне її частиною, яка називається супервізором введення-виведення. Від виконує наступні функції:

1. Отримання запитів на введення–виведення від прикладних програм та власних модулів ОС. Ці запити перевіряються на коректність. Якщо запит виконаний відповідно до специфікацій та не має помилок, він обробляється. У протилежному випадку, видається відповідне діагностичне повідомлення.

2. Виклик розподілювачів каналів та контролерів, планування введення-виведення. Він визначає чергу надання пристроїв задачам або процесам. Їх запити виконуються або ставляться у чергу на виконання.

3. Ініціює операції введення-виведення (передає керування відповідним драйверам) і у випадку, коли керування введенням-виведенням використовує переривання, надає процесор диспетчеру задач для надання його задачі, що стоїть першою у черзі на виконання.

4. При отриманні сигналів переривання від пристроїв введення-виведення ідентифікує їх та передає керування відповідній задачі обробки переривань.

5. Виконує передачу повідомлень про похибки введення-виведення.

6. Надсилає повідомлення про завершення операції введення-виведення процесу, що її запитав, та знімає його зі стану очікування введення-виведення, якщо процес очікував її завершення.

Якщо пристрій є ініціативним (це зовнішні пристрої, окрім стандартних пристроїв введення-виведення, наприклад, датчики), керування з боку супервізора введення-виведення буде полягати у активізації відповідного обчислювального процесу (або переведення його у стан готовності).

Тобто, прикладні програми безпосередньо не зв'язуються з пристроями введення-виведення, незалежно від виду використання – монопольного чи сумісного. Вони це роблять централізовано, через використання супервізора введення-виведення.

8.3 Режими керування введенням-виведенням

Існують два основних режими введення-виведення:

1) режим обміну з опитуванням готовності пристрою введення-виведення;

2) режим обміну з привілеями.

Нехай керування виконує центральний процесор (рис.8.1). Тобто використовується програмний канал обміну між зовнішнім пристроєм та оперативною пам'яттю (ще можливий обмін між зовнішнім пристроєм та каналом прямого доступу у пам'ять за допомогою спеціального з’єднання). Центральний процесор надає пристрою керування команду на виконання введення-виведення. Останній виконує команду, перетворюючи її у послідовність сигналів, зрозумілих пристрою введення-виведення.

Але швидкодія пристрою введення-виведення на декілька порядків менше ніж у процесора. Тому сигнал щодо прийняття чергової команди треба досить довго чекати, постійно опитуючи відповідну лінію інтерфейсу на проходження відповідного сигналу. У режимі опитування готовності, драйвер, що керує процесом обміну, дає виконати команду «перевірити

наявність сигналу готовності». До тих пір, доки останній не з’явиться, даремно витрачається час процесора.

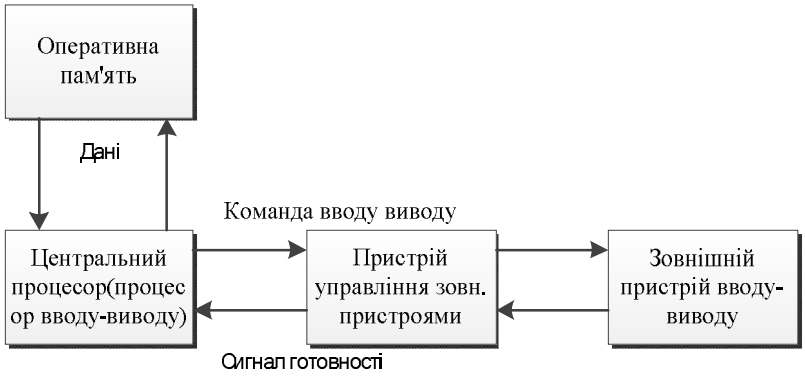


Рис. 8.1. Керування введенням-виведенням

Значно вигідніше тимчасово забути про пристрій введення-виведення після видачі команди введення-виведення. Саме ці сигнали готовності і будуть сигналами запиту на переривання.

Режим обміну з перериваннями у режимі асинхронного керування. Для того, щоб не загубити зв'язок з пристроєм після надання процесом чергової команди з керування обміном і переходу на виконання інших програм, можна організувати відлік часу, за який пристрій має виконати команду та видати сигнал переривання. Максимальне значення такого часу називають **установкою тайм-аута**. Якщо цей час витрачено, то роблять висновок, що зв'язок із пристроєм втрачено, і керування неможливе, про що користувач або задача отримують відповідне повідомлення.

Драйвери, що працюють у режимі переривань, являють собою комплекс програмних модулів, що мають декілька секцій: секцію запуску, одну або декілька секцій продовження та секцію завершення.

Секція запуску ініціалізує операцію введення-виведення. Вона вмикає пристрій введення-виведення або ініціалізує чергову ітерацію введення-виведення.

Секція продовження виконує основну роботу з передачі даних. Вона і є основним обробником переривання. У загальному випадку, використання інтерфейсу може вимагати декількох послідовностей команд керування, а

сигнал переривання з пристрою, як правило, тільки один. Тому, після виконання чергової секції переривання супервізор переривань при черговому сигналі готовності повинен передати керування черговій секції. Якщо ж така секція тільки одна, вона сама передає керування певному модулю обробки.

Секція завершення виключає пристрій введення-виведення або просто завершує операцію.

Програми, що керують введенням-виведенням у режимі переривань, створювати досить складно. Тому, наприклад, драйвер для друку через паралельний порт в ОС Windows працює не в режимі переривань, а в режимі запитів готовності, що призводить до 100% завантаження процесора. Виконання інших завдань виконується винятково за рахунок витісняючих стратегій, коли процесор примусово передається для виконання інших задач.

8.4 Закріплення пристроїв. Загальні пристрої введення-виведення

Цілий ряд пристроїв не допускає паралельного використання. Якщо такі пристрої закріпити за певним процесом, то може з'ясуватись, що деякі процеси взагалі виконуватись не зможуть. Для організації паралельного використання таких пристроїв вводять так звані віртуальні пристрої. До цього ж має відношення поняття спулінга, тобто імітації паралельного розподілу пристрою введення-виведення з послідовною прогамою.

У цьому випадку, кожному процесу надається, наприклад для друку, не реальний, а віртуальний принтер, а потік символів, що виводяться, направляється не на пристрій друку, а у спеціальні спул-файли на диску. По закінченню віртуального друку, можна вивести цей файл на пристрій для друку. Системний процес, що керує спул-файлом, має назву **спулер**.

8.5 Основні системні таблиці введення-виведення

Кожна ОС має свої таблиці введення-виведення. Їх кількість та склад можуть дуже різнитися. Існують ОС, у яких замість таблиць використовують списки, але використання таблиць призводить до більшої продуктивності.

Базуючись на принципі керування введенням-виведенням через супервізор та використанні драйверами механізму переривань, можна зробити висновок про необхідність створення хоча б трьох системних таблиць.

Перша таблиця зберігає інформацію про всі пристрої введення-виведення, що підключені до даної системи. Вона носить назву **таблиці обладнання**. Кожний елемент такої таблиці має назву UCB (Unit Control Block). UCB, як правило, містить наступну інформацію:

- тип пристрою, конкретна модель, символьне ім’я та характеристика пристрою;
- через який інтерфейс підключено, до якого з’єднання, які порти та лінії запиту переривань використовуються;
- номер та адреса каналу, якщо вони використовуються для керування;
- вказівка на драйвер, який має керувати цим пристроєм, адреса секції запуску та секцій проводження;
- інформація про те, чи використовується буферизація при обміні даними з цим пристроєм; ім’я буферу, якщо він виділений у області пам'яті, що займає програма;
- установка тайм-аута та комірка для лічильника тайм-аута;
- стан пристрою;
- поле вказівника для чергу задач, що очікують пристрій; можливо ще деякі дані.

Оскільки драйвери можуть мати властивості реєнтерабельності (один екземпляр програми може забезпечувати паралельне обслуговування декількох пристроїв одного типу), то у елементі UCB повинна зберігатись

інформація про стан даного пристрою та самі змінні для реєнтерабельної обробки, або вказівку на адресу, де цю інформацію можна відшукати.

Дуже важливим компонентом UCB є вказівник на дескриптор тієї задачі, яка на даний час використовує пристрій введення-виведення. Якщо пристрій вільний – відповідне поле вказівника буде мати нульове значення. Якщо пристрій введення-виведення вже зайнятий, і відповідно вказівник не нульовий, то нові запити до пристрою фіксуються через створення списку дескрипторів тих задач, що очікують даний пристрій.

Друга таблиця визначена для реалізації принципу візуалізації. Бажано, щоб програміст міг враховувати тільки найбільш загальні можливості даного класу пристроїв введення-виведення. Наприклад, принтер має виводити на друк символи або графічні зображення. А накопичувач на дисках мав би використовуватись через файлову систему. Тому запити на введення-виведення програми містять логічне ім’я пристрою, абстрагуючись від його конкретних особливостей та характеристик.

Конкретний пристрій, який відповідає віртуальному (логічному), обирає супервізор за допомогою цієї другої таблиці. Вона отримала назву – таблиця опису віртуальних пристроїв (DRT, device reference table). Тобто її призначення – встановлення зв'язку між віртуальними (логічними) пристроями та реальними пристроями, описи яких задані у першій таблиці. Друга таблиця власне дозволяє супервізору перенаправляти запит на введення-виведення із прикладної задачі на ті програмні модулі та структури даних, які зберігаються у відповідному елементі першої таблиці.

У багатокористувацьких системах така таблиця не одна, а декілька – по одній на кожного користувача.

Третя таблиця необхідна для організації зворотного зв'язку між центральною частиною та пристроями введення-виведення. Це таблиця **переривань.** Вона вказує для кожного запиту на переривання той елемент UCB, який співставляється даному пристрою, підключеному таким чином,

щоб він використовав дану лінію (сигнал) переривання. Взаємозв'язок між таблицями наведено на рис. 8. 2.

Запит на операцію введення-виведення від прикладної програми (1) надходить на супервізор (рис. 8.3). Він перевіряє системний виклик на відповідність специфікації та, у випадку похибки, повертає задачі відповідне повідомлення (1-1). Якщо запит коректний, то він відправляється на супервізор введення-виведення (2).

Супервізор введення-виведення за логічним ім’ям і за допомогою таблиці логічних імен знаходить елемент UCB у таблиці обладнання.

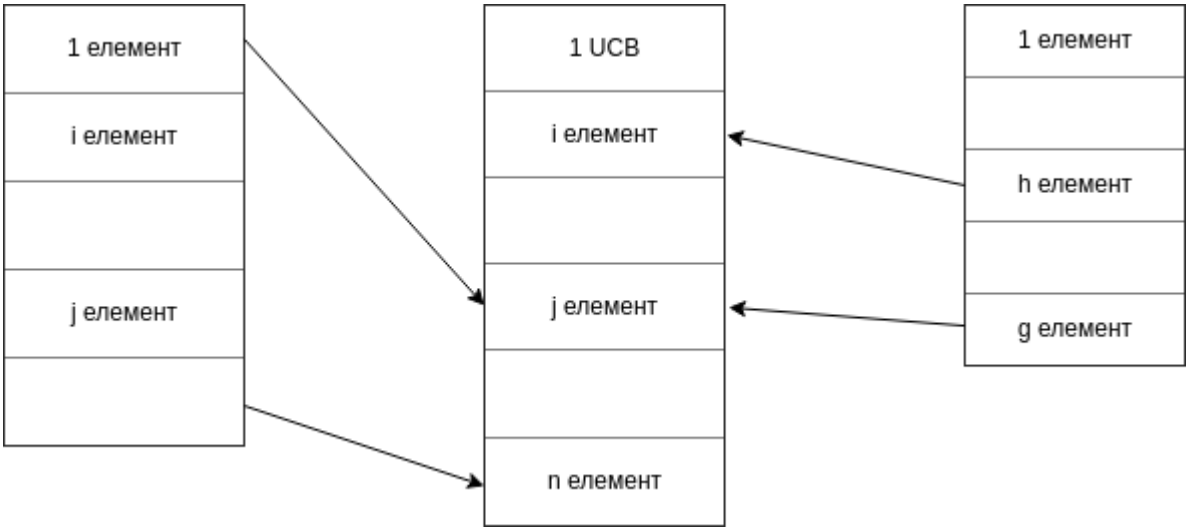


Рис. 8.2. Взаємозв’язок між таблицями введення-виведення

Якщо пристрій зайнятий, то описувач задачі, запит якої зараз опрацьовує супервізор введення-виведення, включається у список задач, що очікують даний пристрій введення-виведення.

Якщо пристрій введення-виведення вільний, супервізор введення-виведення визначає з UCB тип пристрою та при необхідності запускає препроцесор, за допомогою якого отримується послідовність управляючих кодів та даних, яку може правильно сприйняти та відпрацювати пристрій (3). Коли програма управління операцією введення-виведення буде готова, супервізор введення-виведення передає керування відповідному драйверу на секцію запуску (4). Драйвер ініціалізує операцію керування, обнуляє

лічильник тайм-аута і повертає керування супервізору (диспетчеру задач) для того, щоб він встановив на процесор готову для виконання задачу (5).

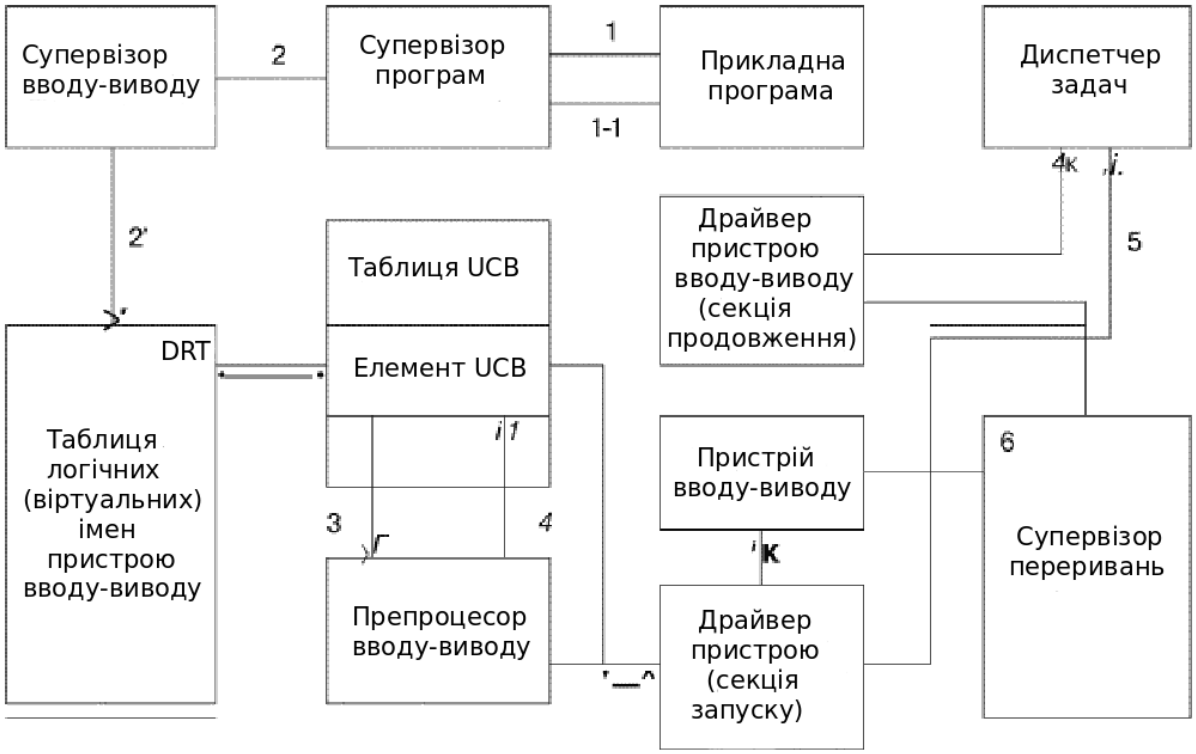


Рис. 8.3. Керування введенням-виведенням

Система продовжує працювати, але коли пристрій введення-виведення відпрацює відправлену йому команду, вона виставить сигнал запиту на переривання, за яким через таблицю переривань управління буде передано на секцію продовження драйверу (6). Отримавши нову команду команду, пристрій знову починає її опрацьовувати, а керування процесом передається диспетчеру задач, і процесор продовжує корисну роботу. Тобто виконується паралельне виконання процесів на фоні виконання введення-виведення.

8.6 Синхронне та асинхронне введення-виведення

Задача, що видала запит на операцію введення-виведення переводиться супервізором у стан очікування завершення цієї операції. Коли супервізор

отримує сигнал від секції завершення про завершенння операції, він переводить задачу (процес) у стан готовності до виконання, і їй може бути надано процесор. Це так зване синхронне введення-виведення. Воно використовується у більшості ОС. Для збільшення швидкодії виконання процесів використовують асинхронний принцип введення-виведення.

Найпростіша реалізація цього принципу полягає у буферизації виводу. Тобто інформація для виведення передається не на пристрій безпосередньо, а у спеціальний системний буфер. У цьому випадку, логічна операція виведення для процесу вважається одразу виконаною, і процес (задача) може не чекати його фактичного виконання. Процесом реального виведенняу, у цьому разі, із системного буфера займається супервізор введення-виведення.

Виділенням буферу із системної області пам’яті займається спеціальний системний процес за вказівкою супервізора введення-виведення.

У результаті, для останнього випадку вивід буде асинхронним, якщо запит на введення-виведення мав вказівку щодо необхідності буферизації, та відповідний пристрій введення-виведення допускає такі асинхронні операції, про які вказано у таблиці UCB.

Введення буферизації як засобу інформаційної взаємодії висуває додаткову проблему керування цими системними буферами. Вона вирішується засобами супервізорної частини ОС. При цьому супервізор вирішує не тільки задачі по виділенню та звільненню буферів у системній області пам'яті, але й задачі синхронізації процесів по заповненню та звільненню буферів, а також побудову черг при відсутності вільних буферів. Як правило, супервізор введення-виведення при цьому використовує ті стандартні засоби синхронізації, які використовує ОС для вирішення інших задач синхронізації.

Керування та буферизація при роботі з магнітними дисками

Середня швидкість процесора з оперативною пам'яттю на 2-3 порядки вища ніж середня швидкість передачі даних із зовнішньої пам'яті на

магнітних дисках в оперативну пам'ять. Для подолання такої невідповідності у продуктивності використовується буферизація та кешування даних.

Найпростіший варіант прискорення дискових операцій читання даних – використання подвійної буферизації. При цьому, доки в один буфер записується інформація, із другого вона читається та передається за запитом. Аналогічно, при записі. Буферизація використовується практично в усіх операційних системах.

Кешування дуже корисне у тому випадку, коли програма (процес) багаторазово читає з диску одні й тіж самі дані. Після того, як вони один раз були розміщені у кеші, звернень до диску більше не треба, і швидкодія суттєво підвищується. У даному випадку, під кешем можна розуміти деякий пул буферів, якими керує системний процес. Якщо запитуємо деяку кількість секторів, то відповідна інформація, проходячи через кеш, там і залишається. Якщо буде потрібне повторне читання, то дані можуть бути прочитані безпосередньо з оперативної пам'яті без звертання до диску.

Кількість буферів, що складають кеш, обмежена, тому виникають ситуації, коли заново прочитані або записані сектори мають бути замінені у цих буферах. Власне, принцип роботи такої кеш пам'яті не відрізняється від роботи звичайної кеш пам'яті., яку розглядали раніше. Процес кешування, як і вказувалось раніше, – це спосіб сумісного використання двох запам’ятовуючих пристроїв із різною швидкодією.

Тому, як і звичайно, кешування дискових операцій може бути покращено за рахунок техніки упередженого читання, коли з диску зчитується значно більша кількість даних, ніж операція запрошувала.

У ряді ОС є можливість вказати у явному вигляді параметри кешування. У деяких – за це відповідає сама ОС. Так при використанні Windows NT такої можливості немає, а у Windows 95/98 є. Можна вказати обсяг пам'яті, що відводиться для кешування та об’єм порції даних (буфер або chunk), з яких набирається кеш. У файлі System.ini у секції [VCACHE] можна прописати, наприклад:

```
[VCACHE]
Min FileCache = 4096
Max FileCache = 32768
ChunkSize = 512
```

Мінімальний обсяг кешу – 4 Мбайт, максимальний обсяг – 32 Мбайт, об’єм буфера, яким оперує менеджер кешу, дорівнює об’єму одного сектора, тобто 512 байт.

У мультипрограмних системах при виконанні багатьох задач запити на читання та запис даних надходять таким потоком, що створюються черги. Якщо обслуговувати ціі черги у порядку надходження у чергу, то, завдяки випадковому характеру звертань до тих чи інших секторів, мають місце великі втрати часу на пошук. Тому, враховуючи те, що переупорядкування черги з метою зменшення непродуктивних втрат часу можливо виконувати відносно швидко, можна запропонувати деякі дисципліни обслуговування таких черг з метою економії непродуктивного використання часу. Наведемо деякі із них.

SSTF (Shortest Seek Time First) найменший час пошуку – перший. У відповідності з цією дисципліною? задовольняється запит, що потребує мінімальної кількості кроків для перепозиціонування записуючих голівок на диску.

SCAN (сканування). Згідно цієї дисципліни голівки зчитувача переміщуються то в одному, то в другому напряму, обслуговуючи на своєму шляху усіх, хто виставив вимоги і потрапляє на такому шляху.

NEXT Step Scan – відмінність від попередньої дисципліни у тому, що обслуговуються тільки ті запити, які вже існували на момент початку переміщення.

C-Scan (циклічне обслуговування). Відповідно до цієї дисципліни головки переміщуються циклічно з зовнішньої до внутрішньої поверхні, обслуговуючи усі запити, що трапляються на шляху такого переміщення.

8.7 Організація доступу до зовнішних пристроїв

З точки зору центрального процесора, зовнішні пристрої – це набір спеціалізованих комірок пам’яті, або пристрої є наборами спеціалізованих елементів пам'яті або, якщо завгодно, регістрів. У процесорів загального призначення регістри пристроїв підключаються до адресних шин та шин даних. Пристрій має адресний дешифратор. Якщо виставлена на шині адреса відповідає адресі одного з регістрів пристрою, дешифратор підключає відповідний регістр до шини даних (рис. 8.4). Таким чином, регістри пристрою отримують адреси у фізичному адресному просторі процесора. Існують два підходи до адресації цих регістрів:

- окремий адресний простір введення-виведення,
- відображення у пам’яті введення-виведення (memory maped I/O).

У першому випадку, для звернення до регістрів пристроїв використовуються спеціальні команди, наприклад**, in** та **out.**

У другому випадку, пам’ять та регістри зовнішних пристроїв розміщені у одному адресному просторі, і можна використовувати які завгодно команди, що дозволяють працювати з операндами у пам’яті.

Як правило, навіть у випадку різних адресних просторів, для обміну даними між пам’яттю та зовнішними пристроями процессор використовує одні й ті ж самі шини адрес та даних, але має додатковий сигнал адресної шини, що вказує на те, який з адресних просторів використовується у конкретному випадку.

Існують два основні підходи до виділення адрес зовнішнім пристроям: **фіксована адресація,** коли один і той самий пристрій завжди має одні й ті ж самі адреси регістрів, і **географічна адресація**, коли кожному роз'єднувачу периферійної (або системної шини) відповідає свій діапазон адрес.

Географічно можна розподіляти не лише адреси регістрів, але й інші ресурси: лінії запиту переривань, канали прямого доступу до пам’яті і т. д.

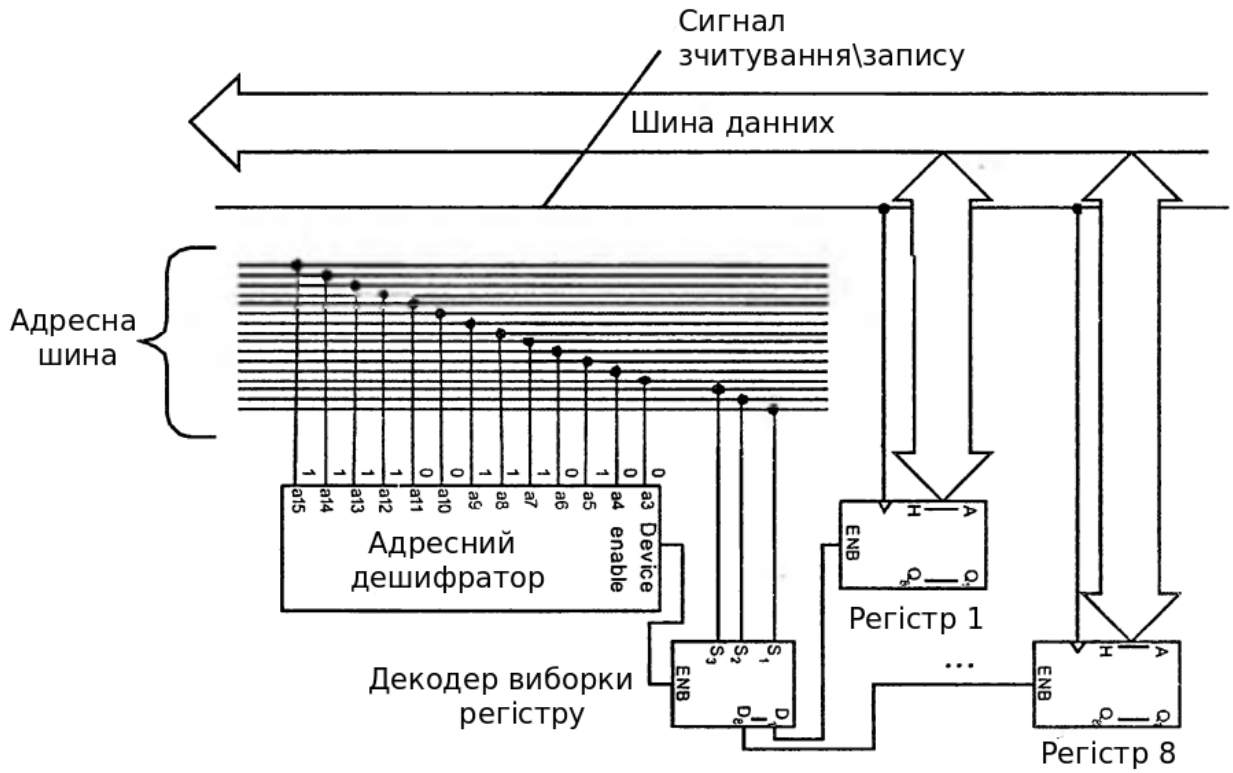


Рис. 8.4. Підключення зовнішнього пристрою до шини

Цей спосіб розподілу адресного простору зручний тим. що унеможливлює конфлікти адрес між пристроями різних виробників або між двома однотипними пристроями (з цією проблемою має бути знайомий кожен, хто намагався одночасно встановити в комп'ютер мережеву і звукову карти конструктиву ISA). Більшість **периферійних шин** сучасних міні- і мікрокомп'ютерів, такі, як PCI, S-Bus і ін., реалізують географічну адресацію.

Багато сучасних конструктивів вимагають, аби окрім регістрів управління і даних, пристрої мали також конфігураційні регістри, через звернення до яких ОС може отримати інформацію про пристрій: фірму-виробника, модель, версію, кількість регістрів тощо. Наявність таких регістрів дозволяє ОС без втручання (або з мінімальним втручанням) з боку адміністратора визначити встановлене в системі устаткування і автоматично підвантажити відповідні модулі управління.

8.8 Порти введення-виведення

Цей пристрій – стандартний компонент більшості мікропроцесорних систем.

Порт виведення представляє собою регістр з входами і виходами (рис. 8.5). Кількість виходів порту відповідає кількості бітів регістру. Нулю у розряді відповідає низький рівень напруги (скажімо, 0), а одиниці – високий рівень (скажімо, напруги блоку живлення).

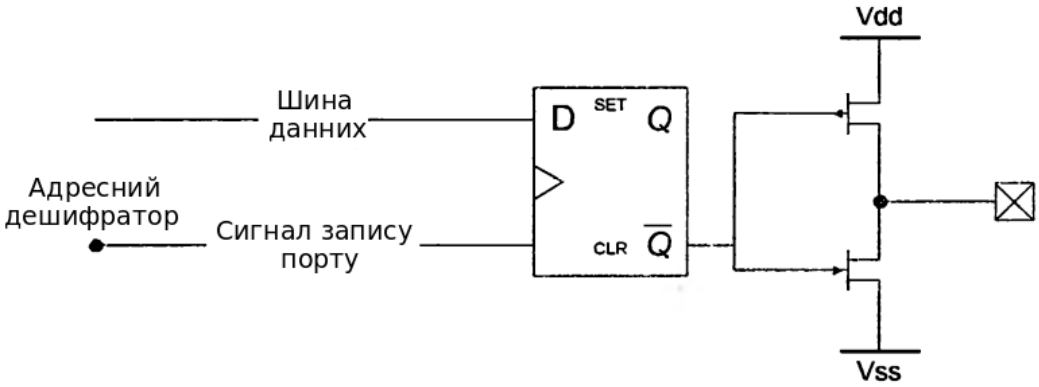


Рис. 8.5. Структура порту виведення

Порт введення теж складається з регістру та декількох вхідних ліній, що відповідають бітам регістра (рис. 8.6). Біт регістра має значення 0, коли на вхід подають низьку напругу і, навпаки, одиниці – високу. Регістр зберігає прийнятий код.

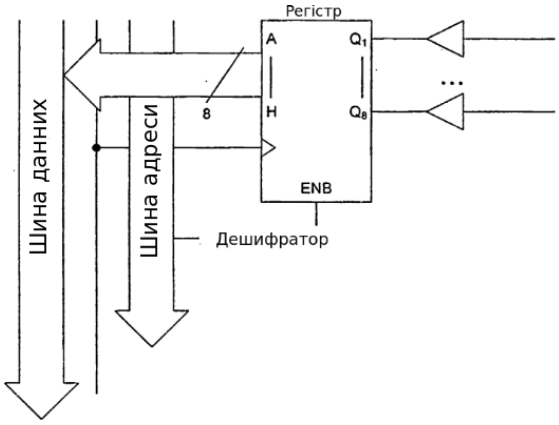


Рис. 8.6. Структура порту введення

Розробники мікропроцесорів часто роблять сумісними порти введення-виведення. Головна проблема використання простих портів полягає у тому, що приймаючий пристрій має знати, чи передавач виставив на своїх шинах нову порцію даних, чи ні. Для рішення цієї проблеми є три підходи:

- синхронна передача даних,
- асинхронна передача даних,
- ізохронна передача.

При **синхронній передачі** або передаємо додатковий сигнал – строб, або передаємо синхросигнали тими самими лініями передачі сигналу, що й дані.

Передача стробу потребує прокладки додаткових дротів, але можливо суміщення синхросигналу та даних, і тому це широко використовується.

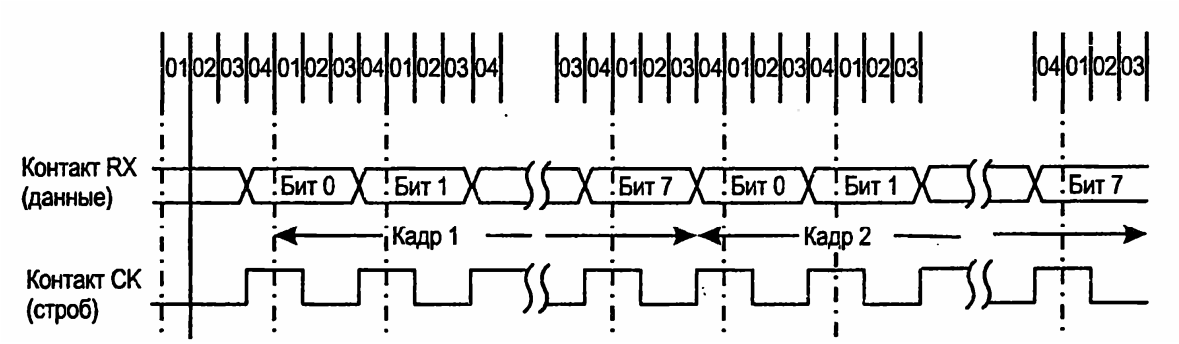


Рис. 8.7. Часова діаграма порту зі стробом

Більшість портів із стробом асиметричні: один з пристроїв є **ведучим** (master – хазяїн), і генерує стробовий сигнал, а другий –**ведений** (slave – раб), що користується цим сигналом для прийому чи передачі.

Часто замість одиничного стробу використовується декілька різних сигналів. Наприклад, один виставляється передавачем та сповіщає, що чергова порція даних готова, а другий сигнал – приймачем і сповіщає, що приймач прийняв ці дані та готовий прийняти чергові.

Додаткові сигнали також можуть вирішити питання про те, яке з пристроїв у даному циклі буде приймачем, а яке – передавачем.

При **асинхронному** обміні даними пристрій – передавач пересилає стартовий символ, сигналізуючий про те, що зараз підуть дані, та із

фіксованим інтервалом виставляє на своїх виходах сигнали даних. Переданий за один раз блок даних, звичайно невеликий за обсягом з причин можливої небезпеки від того, що годинники приймача та передавача, за якими вони відміряють свої інтервали між черговими порціями даних, є неточними та можуть не співпадати.

Звичайно блок даних складається із фіксованої кількості символів і називається кадром або фреймом (frame). Кадр завершується одним або декількома стоповими символами. Не знайшовши цих символів, приймач може зрозуміти, що його годинник розійшовся з годинником передавача.

Асинхронна передача дозволяє економити на дротах за рахунок стропових сигналів та при цьому запобігти складних способів кодування, характерних для суміщеної синхронної передачі. Однак стартові та стопові символи складають значну частку потоку даних, що значно збільшує накладні витрати. Окрім цього, при передачі великого об’єму даних у вигляді тісно слідуючих один за одним кадрів, виникає небезпека, що приймач загубить заголовок чергового кадру та не зможе поновити структуру потоку. Щоб запобігти цьому, багато асинхронних протоколів потребують паузи між сусідніми кадрами.

Асинхронна передача кадрів доцільна у ситуаціях, коли обсяги передачі невеликі, а потреба у їх передачі виникає у випадкові, непередбачувані моменти часу.

Звичайно асинхронні порти працюють з невисокою швидкодією – не більше декількох кілобіт на секунду.

Ізохорна передача даних нагадує асинхронну з тією різницею, що при обміні даними приймач і передавач користуються високо стабільними, незалежними тактовими генераторами. Тому вони можуть обмінюватися кадрами великого розміру. В ідеалі ізохорна передача дуже якісна, але дуже складно забезпечити стабільність та калібровку тактових генераторів, тому вона використовується досить рідко.

Як синхронні, так і асинхронні порти бувають наступних типів (рис. 8.8):

- симплексні (simplex) – тільки один пристрій може передавати дані;

- напівдуплексні (half-duplex) – обидва пристрої можуть приймати та передавати дані, але неспроможні робити це одночасно, наприклад, оскільки прийом та передача використовують один дріт;

- повнодуплексні (full-duplex) або просто дуплексні – обидва пристрої можуть одночасно передавати та приймати дані, найчастіше різними дротами.

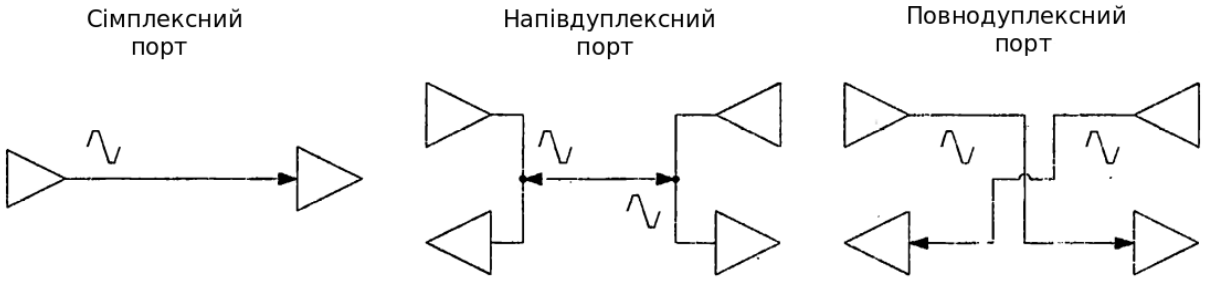


Рис. 8.8. Різновиди портів

Ще одна класифікація портів передачі даних (рис. 8.9):

- послідовні порти;
- паралельні порти.

Послідовний порт має один дріт, яким послідовно передаються біти даних, а також можливо синхронізуючі або стартові і стопові біти.

Паралельний порт має декілька ліній передачі даних, як правило 8, для передачі одного байта, а іноді й більше. Як правило, послідовні порти – асинхронні або синхронні із суміщеною передачею синхросигналу. Паралельні порти – синхронні, зі стробами. До восьми шин додається ще одна – дев`ята для передачі строба. Тож додаткові витрати у процентному відношенні для цього випадку невеликі.

Розглянемо порт RS232, який широко використовується для передачі даних назовні комп`ютера. RS232 використовує як 1 напругу у діапазоні від -25 до -3 В, а як 0 – відповідно, від +3 до +25В.

RS232 передбачає двосторонній обмін даними. Для цього передбачені дві лінії даних – для прийому та передачі, які звичайно позначають як Tx та Rx

У відповідності із стандартом, пристрої поділяють на два типи: комп`ютери та термінали. Різниця між ними полягає у тому, що комп`ютер передає дані лінією Tx, а отримує їх по Rx. Термінали – навпаки. Для з'єднання двох комп`ютерів необхідно мати спеціальний, так званий нуль – модемний кабель, у якому дроти Tx та Rx перехрещені.

Послідовний порт

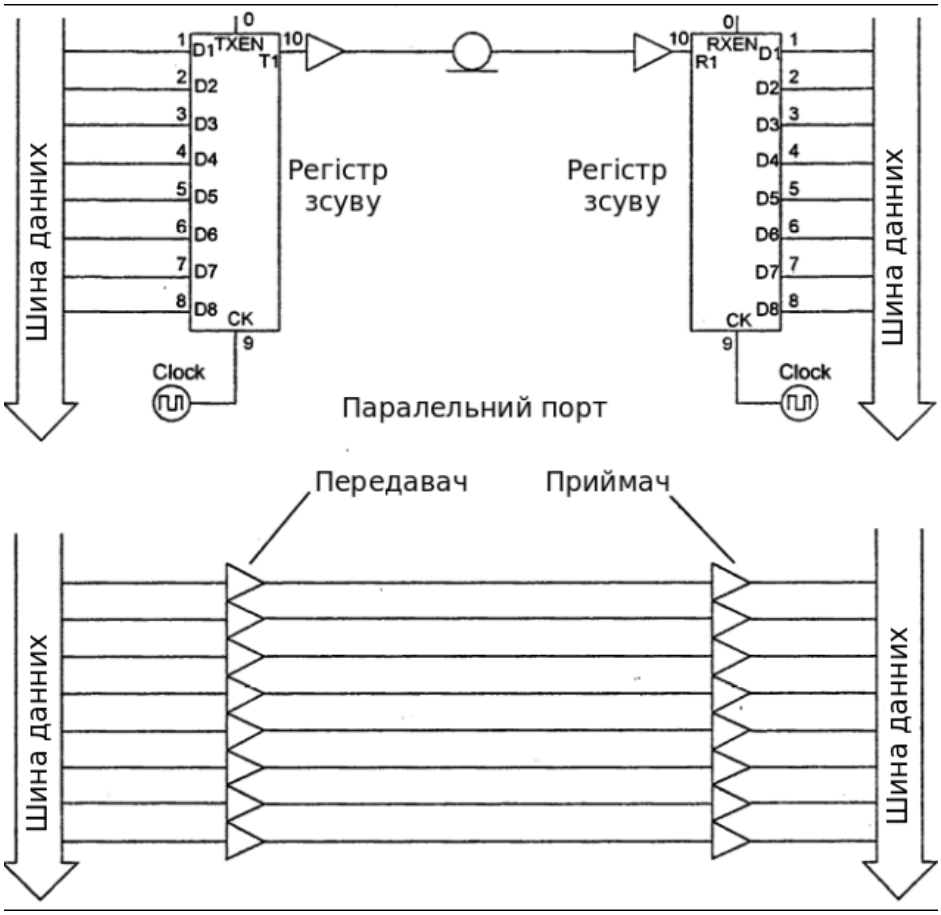


Рис. 8.9. Послідовний та паралельний порти

Обмін даними виконується кадрами, що складаються із стартового біта, семи чи восьми бітів даних (молодший біт передається першим), можливого контрольного біта парності та одного чи двох стопових бітів (рис. 8.10).

Мінімальна швидкість передачі 300 біт/с, максимальна – більше 115200 біт/с. Швидкість та варіації формату кадру визначаються налаштуваннями приймача та передавача. Необхідно, щоб у з'єднаних портом пристроїв вони співпадали.

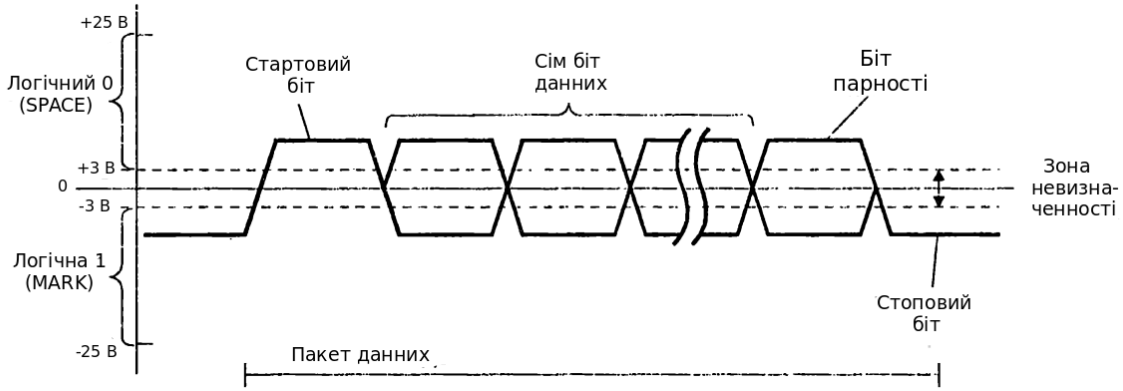


Рис. 8.10. Діаграма напруг

8.9 Шини

Часто виникає потреба у підключенні до одного порту передачі даних декількох пристроїв. При цьому кожна порція переданих даних має супроводжуватись вказівкою, для якого з цих пристроїв вона призначена, – адресою або селектором пристрою. Такі багатоточкові порти (рис. 8.11) звуться **шинами** (bus).

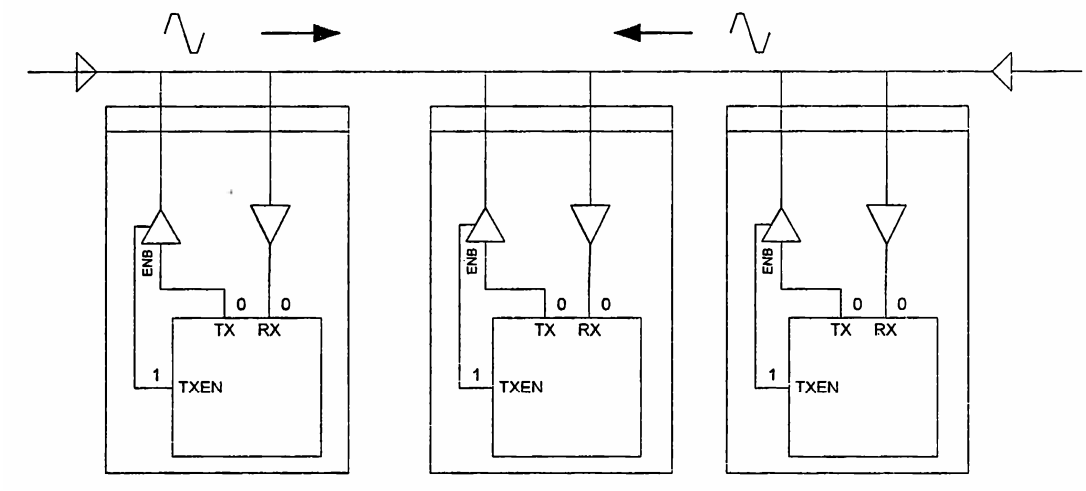


Рис. 8.11. Шина

Як і двоточкові порти, шини бувають синхронні та асинхронні, а також послідовні та паралельні. Однак, терміни синхронний та асинхронний використовуються у іншому визначенні, ніж при описі портів. Асинхронними називають шини, у яких відений пристрій не виставляє сигнали завершення операції, а синхронними, відповідно, шини, де відений зобов`язаний це робити.

Підключення N пристроїв двоточковими портами вимагає, щоб центральний процесор мав N прийомопередатчиків. Використання багатоточкового порту дозволяє обходитись одним. За рахунок цього, виникає можливість зменшити кількість виходів мікросхеми процесора або периферійного контролера. Окрім цього, при вдалому розміщенні пристроїв можна отримати значний виграш у загальній кількості дротів та зменшити кількість доріжок на друкованій платі. У багатьох випадках, це призводить до здешевлення апаратури, хоча і ускладнює протокол передачі даних.

Основний недолік шини полягає у тому, що тільки один пристрій на шині може передавати інформацію у кожен момент часу. Якщо у двоточкових портах часто доцільно реалізувати повнодуплексний обмін даними за допомогою двох комплектів ліній (один на прийом, інший – на передачу, як у випадку RS232), то, у випадку шинної топології, це неможливо. Тому шини бувають тільки **напівдуплексні** або **симплексні.**

Неможливість паралельно виконувати обмін з двома пристроями може призвести до падіння продуктивності порівняно з власним портом обміну для кожного пристрою. Але якщо пристрої не використовують пропускну здатність каналів передачі даних повністю, програш виявляється не таким вже великим. Тому шини широко використовують навіть тоді, коли один пристрій має можливість ініціювати обмін даними.

Якщо передачу даних можуть ініціювати декілька пристроїв (наприклад, системна шина з декількома процесорами), шинна технологія найбільш прийнятна. Така технологія, однак, потребує вирішення однієї проблеми: забезпечення арбітражного доступу до шини з боку можливих ініціаторів обміну.

Методи такого арбітражу відрізняються великою різноманітністю. Необхідність визначення колізій та арбітражу накладає обмеження на фізичні розміри шини. Сигнали розповсюджуються шиною з кінцевою швидкістю, а будь-які два конфліктуючі пристрої мають дізнатися про колізію за час, що менший, ніж мінімальний цикл передачі даних. Для внутрішніх шин мікропроцесора ця проблема може бути неактуальною, однак для локальних мереж, довжина яких може становити метри і кілометри, чи для системних шин комп`ютерів з великою кількістю процесорів та банків пам`яті – це серйозна проблема. Одним з основних шляхів подолання цієї проблеми – заміна шини центральним комутатором або системою комутаторів. Пристрої з`єднуються з найближчим комутатором повнодуплексним двоточковим каналом, канали також поєднують комутатори один з одним, а усі колізії виникають та вирішуються тільки усередині комутаторів.

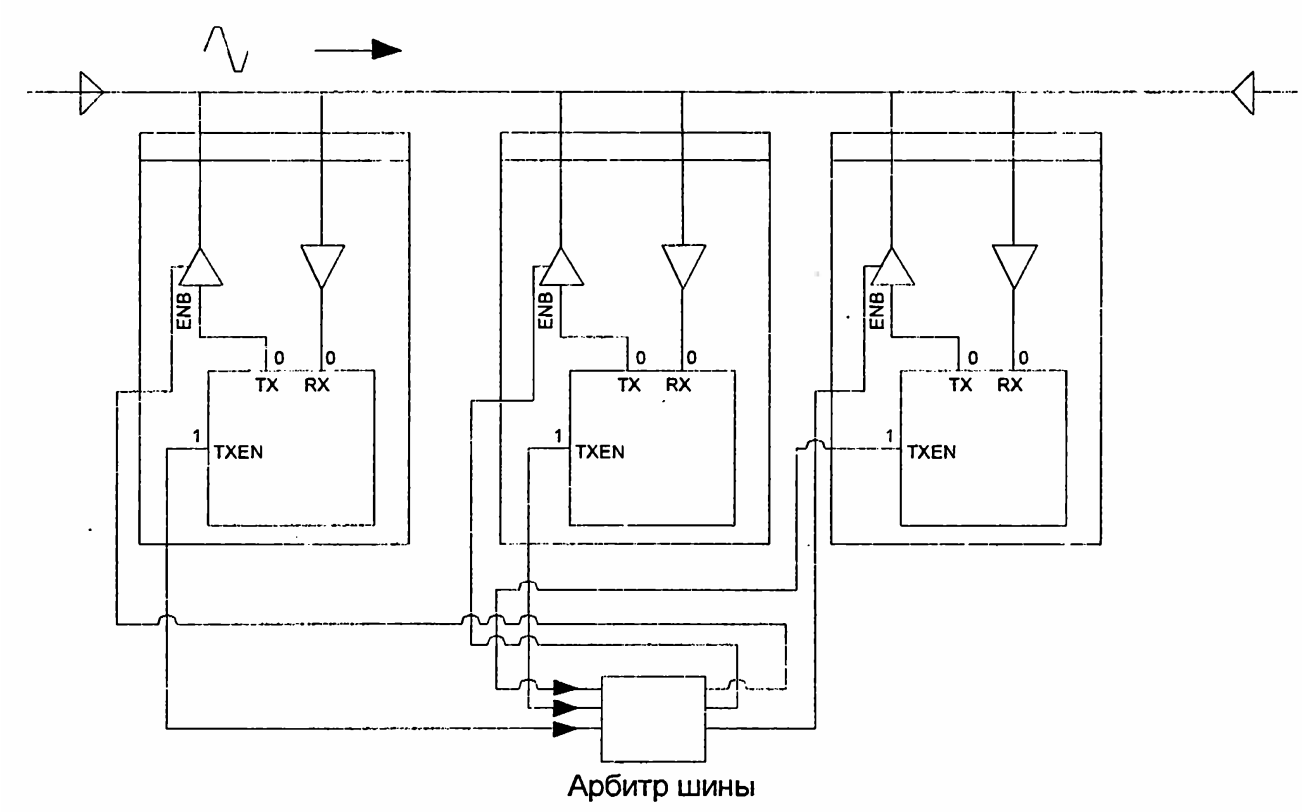


Рис. 8.12. Шина з декількома передатчиками

Внутрішня шина комутатора має велику пропускну здатність (значно більшу, ніж зовнішні з`єднання), окрім того, комутатори, як правило, мають декілька внутрішніх шин. Тому вирішення колізій всередині комутатора часто зводиться до відправки даних іншим шляхом (рис. 8.13).

Таким чином, магістраль, що комутується, дозволяє зменшити втрати продуктивності, що виникає завдяки колізіям, та підвищити реальну пропускну здатність магістралей. Однак, при цьому суттєво збільшується загальна вартість системи, що далеко не завжди може бути виправдано. У обчислювальних системах загального призначення часто використовується топологія з двома або більше шинами. Процесори та пам`ять зв`язані системною шиною. До цієї шини також прив`язують декілька адаптерів периферійних шин, до яких підключають зовнішні пристрої. Головні переваги такого рішення полягають у тому, що одні й ті ж самі периферійні пристрої можуть бути використані у обчислювальних системах з різними системними шинами. Більш детальна інформація за темою зовнішніх

пристроїв, каналів зв`язку та драйверів зовнішніх пристроїв може бути отримана з рекомендованої літератури.

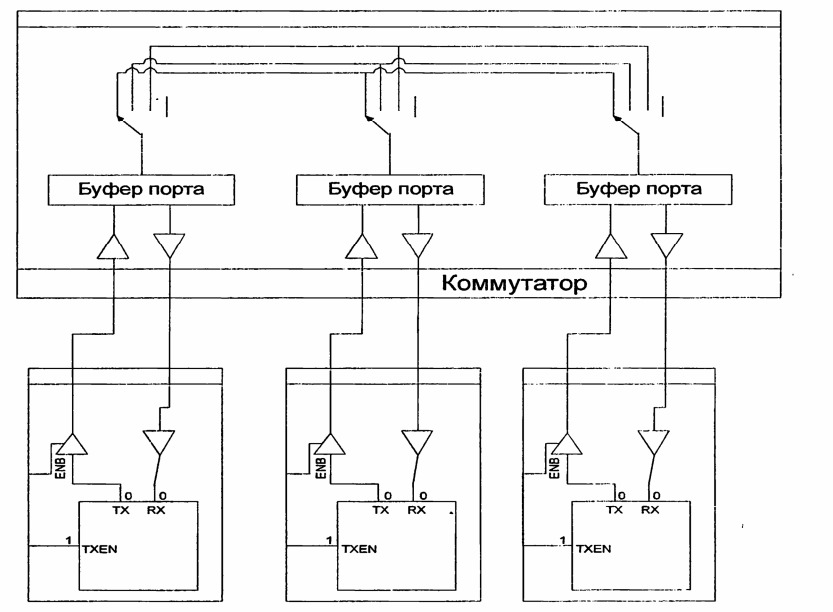


Рис. 8.13. Комутатор з декількома внутрішніми шинами

8.10 Контрольні запитання до розділу 8

1. Які завдання вирішує підсистема введення-виведення?

2.Які існують основні режими введення-виведення?

3. Які завдання виконує спулер?

4. Які існують таблиці введення-виведення і яку вони зберігають інформацію?

5. Яким чином організовано синхронне і асинхронне введення-виведення?

6. Яким чином організовано доступ до зовнішніх пристроїв, що являють собою порти введення-виведення?

7. Яка різниця між асинхронною та ізохронною передачами даних?

8. Які бувають типи портів?

9. Які типи портів називають шинамиі і які можливі типи шин?

Розділ 9. Організація паралельної роботи пристроїв введення-виведення і процесора

Кожен пристрій введення-виведення забезпечується блоком керування – контролером. Контролер взаємодіє із драйвером – системним програмним модулем, що керує даним пристроєм.

Контролер періодично приймає від драйвера виведену на пристрій інформацію й команди керування, які визначають, що робити із цією інформацією.

Пристрій введення-виведення працює під управлінням контролера в проміжках часу між видачею команд незалежно від ОС.

Від підсистем введення-виведення вимагається спланувати в реальному масштабі часу запуск і призупинення драйверів, забезпечивши прийнятний час реакції кожного драйвера на незалежні дії контролера.

Для цього всі драйвери розподіляють на декілька пріоритетних рівнів відповідно до вимог за часом реакції й витрат процесорного часу.

Для реалізації цього процесу звичайно використовується диспетчер переривань ОС.

9.1 Узгодження швидкостей обміну і кешування даних

При обміні даними виникає задача узгодження швидкостей. Вона вирішується за рахунок буферизації даних в оперативній пам’яті (ОП) і синхронізації доступу процесів до буфера.

Але буферизація тільки на основі ОП у підсистемі введення-виведення виявляється недостатньою – різниця між швидкістю обміну з ОП і швидкістю роботи зовнішнього пристрою виявляється занадто великою. З іншого боку, при великих обсягах введення-виведення ОП просто може не вистачити.

Для таких випадків як буфер використовується спул (spool). Типовий приклад – вивід на принтер. Друк документа у кілька десятків мегабайт не є рідкістю.

Інше рішення проблеми – більша буферна пам'ять у контролерах зовнішніх пристроїв. Наприклад, контролери графічних дисплеїв. Їх ОП порівнянна з ОП процесора.

9.2 Розподіл пристроїв і даних між процесами

Пристрої введення-виведення можуть надаватися процесам, як у монопольному, так й у поділюваному режимах.

ОС повинна забезпечувати контроль доступу тими ж способами, що й при доступі процесів до інших ресурсів обчислювача.

ОС може контролювати доступ не тільки до пристрою в цілому, але й до **окремих порцій даних**. Наприклад, при виводі на графічний дисплей – інформація в окремих вікон екрана.

Тому для організації спільного доступу до частин пристрою або частин даних неодмінною умовою є задання режиму спільного використання пристроїв або даних у цілому.

ОС надає пристрої, відслідковуючи процедури захоплення й звільнення пристроїв, оптимізуючи послідовність операцій введення-виведення для різних процесів з метою підвищення загальної продуктивності, якщо це можливо.

Наприклад, при обміні даними декількох процесів з дисками можна впорядкувати послідовність операцій.

9.3 Забезпечення зручного логічного інтерфейсу між пристроями й іншою частиною системи

Всі сучасні ОС підтримують як основу такого інтерфейсу файлову модель периферійних пристроїв. При такому підході будь-який пристрій виглядає як

набір байтів, з якими можна працювати за допомогою уніфікованих системних викликів (наприклад, read, write), задаючи ім'я файлу - пристрою й зсув від початку послідовності байтів.

Привабливість такої моделі полягає в її простоті й уніфікованості для пристроїв різного типу.

Іноді для спеціальних застосувань цей інтерфейс використовується як базовий і вимагає додаткової доробки. Наприклад, при програмуванні операцій мережного обміну або виведенні на дисплей графічної інформації.

9.4 Підтримка широкого спектра драйверів і простота включення нового драйвера в систему

Наявність різноманітних драйверів для всіх типів зовнішніх пристроїв є важливою характеристикою ОС. Наприклад, така гарна у багатьох відношеннях ОС як ОС/2 була витіснута ОС Windows завдяки багатству драйверів.

Інтерфейсу драйверів необхідна відкритість, тобто доступність його опису для незалежних розроблювачів ПО.

Драйвер взаємодіє, з однієї сторони, з модулями ядра ОП (підсистемою введення-виведення, системними викликами, керування процесами й пам'яттю), з іншого боку, з контролерами зовнішніх пристроїв.

Тому існують 2 типи інтерфейсів:

- драйвер - ядро;
- драйвер - пристрій.

Інтерфейс драйвер - ядро повинен бути обов’язково стандартизований.

Інтерфейс драйвер - пристрій має сенс стандартизувати тоді, коли підсистема введення-виведення не дозволяє драйверу безпосередньо взаємодіяти із апаратурою.

Екранування драйвера від апаратних засобів є досить корисною функцією, тому що драйвер стає незалежним від апаратної платформи.

Для підтримки процесу розробки драйверів для ОС додатково випускається пакет DDK (Driver Development Kit), що представляє собою необхідний інструментарій: бібліотеки, компілятори й відладчики.

9.5 Динамічне завантаження і вивантаження драйверів

Включення драйвера до складу модулів працюючої ОС являє собою самостійну проблему.

Оскільки набір потенційно підтримуваних даною ОС периферійних пристроїв завжди ширший ніж набір конкретної системи, то дуже цінною властивістю ОС є можливість динамічно завантажувати в ОП необхідний драйвер (без зупинки ОС) і вивантажувати його після того, як потреба в підтримці пристрою зникає. Це може істотно заощадити системну область пам'яті.

Альтернативою динамічному завантаженню драйверів при змінах поточної конфігурації зовнішніх пристроїв є повторна компіляція коду ядра з необхідним набором драйверів, що створює між усіма компонентами ядра статичні зв'язки замість динамічних. При цьому зміни в процесі роботи ОС неможливі.

9.6 Підтримка декількох файлових систем

Диски – це особливий вид периферійних пристроїв, тому що на них зберігається більша частина користувацьких і системних даних. Ці дані організуються у файлові системи, властивості яких багато в чому визначають властивості ОС (відмовостійкість, швидкодію, максимальний обсяг збережених даних). Хороша файлова система звичайно переноситься з однієї ОС в іншу.

Так файлова система FAT спочатку була розроблена для MS-DOS, потім перейшла в OS/2 й MS Windows .

Важливо, щоб архітектура підсистеми введення-виведення дозволяла досить просто включати у свій склад нові типи файлових систем без необхідності переписування коду.

Для цього в ОС передбачається спеціальний шар ПО. Наприклад, шар VFS (Virtual File System) у версіях UNIX.

9.7 Підтримка синхронних і асинхронних операцій введення–виведення

Операції введення-виведення можуть виконуватися в синхронному й асинхронному режимах.

Синхронний режим означає, що процес, що запросив операцію, припиняє свою роботу доти, поки операція введення-виведення не завершиться.

При асинхронному режимі процес виконується в мультипрограмному режимі одночасно з операцією введення-виведення.

Підсистема введення-виведення надає своїм клієнтам (користувацьким процесам й ядру ОС) можливість виконувати як синхронні, так й асинхронні операції введення-виведення залежно від потреб викликаючої сторони.

Системні виклики введення-виведення найчастіше оформлюються як синхронні процедури у зв'язку з тим, що такі операції тривають довго, й користувацькому процесу або потоку однаково прийдеться чекати результатів, щоб продовжити роботу.

Внутрішні виклики з модулів ядра ОС виконуються як асинхронні процедури, тому що ядру потрібен вільний вибір подальшого поводження після запиту операції вводу - виводу.

9.8 Багатошарова модель підсистеми введення-виведення

При великій розмаїтості пристроїв введення-виведення, що значно відрізняються, необхідно дотримуватись балансу між двома суперечливими вимогами:

- необхідністю обліку всіх особливостей кожного пристрою;

- єдиним логічним поданням й уніфікованим інтерфейсом для пристроїв всіх типів.

Нижні шари підсистеми введення-виведення повинні включати індивідуальні драйвери конкретних фізичних пристроїв, а верхні шари повинні узагальнювати процедури керування цими пристроями, надаючи, якщо можливо, загальний інтерфейс.

9.9 Менеджер введення-виведення

Модулі підсистеми введення-виведення, які організують погоджену роботу всіх компонентів підсистеми й взаємодію з користувацькими процесами й іншими підсистемами ОС, утворюють начебто оболонку підсистеми.

Ця оболонка називається менеджером введення-виведення.

Верхній шар менеджера підтримує **користувальницький інтерфейс вводу–виводу**, тобто приймає запити на введення-виведення і переадресовує їх драйверам, а також повертає процесам результати операцій вводу–виводу.

Нижній шар взаємодіє з контролерами зовнішніх пристроїв, екрануючи драйвери від особливостей апаратури вводу–виводу, системи переривань тощо. Цей шар приймає від драйверів запити на обмін даними з регістрами контролерів у деякій узагальненій формі з використанням незалежних від шини вводу–виводу адресації й формату, перетворюючи ці запити у формат, зрозумілий апаратній платформі.

Ще однією функцією менеджера введення-виведення є організація взаємодії модулів введення-виведення з модулями інших підсистем ОС, таких як підсистеми керування процесами, віртуальною пам'яттю й ін.

Наявність стандартного віртуального міжмодульного інтерфейсу підвищує стійкість і поліпшує розширюваність підсистеми введення-виведення, хоча й сповільнює її роботу.

9.10 Багаторівневі драйвери

Традиційні особливості й функції, виконувані драйвером, полягають у наступному:

- входить до складу ядра ОС, працюючи в привілейованому режимі;
- безпосередньо управляє зовнішнім пристроєм, взаємодіючи з його контролером за допомогою команд введення-виведення комп'ютера;
- обробляє переривання від контролера;
- надає програмістові зручний логічний інтерфейс, екрануючи від непотрібних деталей;
- взаємодіє з іншими модулями.

Традиційні драйвери не ділилися на шари. З розвитком ОС, поряд із традиційними з'явилися **високорівневі драйвери**, які розташовуються над традиційними драйверами. При цьому традиційні драйвери стали називати апаратними. За допомогою **високорівневого** драйвера підвищується гнучкість і розширюваність функцій керування пристроєм – замість жорсткого набору функцій, які зосереджені у низкорівневому драйвері, адміністратор ОС може вибрати необхідний набір функцій, установивши потрібний **високорівневий** драйвер.

Якщо різним додаткам необхідно працювати з різними логічними модулями одного й того ж фізичного пристрою, то для цього достатньо установити в системі на одному рівні кілька драйверів, що працюють над одним апаратним драйвером.

Кількість рівнів драйверів звичайно не обмежується. На практиці використовується від 2 до 5 рівнів.

В**исокорівневі** драйвери оформляються за тими же правилами і дотримуються тих же внутрішніх інтерфейсів, що й апаратні драйвери, за винятком того, що вони не викликаються за перериваннями, тому що взаємодіють з апаратурою.

Як саме загальні принципи побудови багаторівневих драйверів можуть бути реалізовані віповідно до конкретних пристроїв, розглянемо на прикладі керування дисками.

Апаратні драйвери підтримують для верхніх рівнів подання диску як послідовного набору блоків однакового розміру, перетворюючи разом з контролером номер блоку в більш складну адресу з номером циліндра, головки й сектора. Поняття «файлу» і файлової системи апаратні драйвери не підтримують.

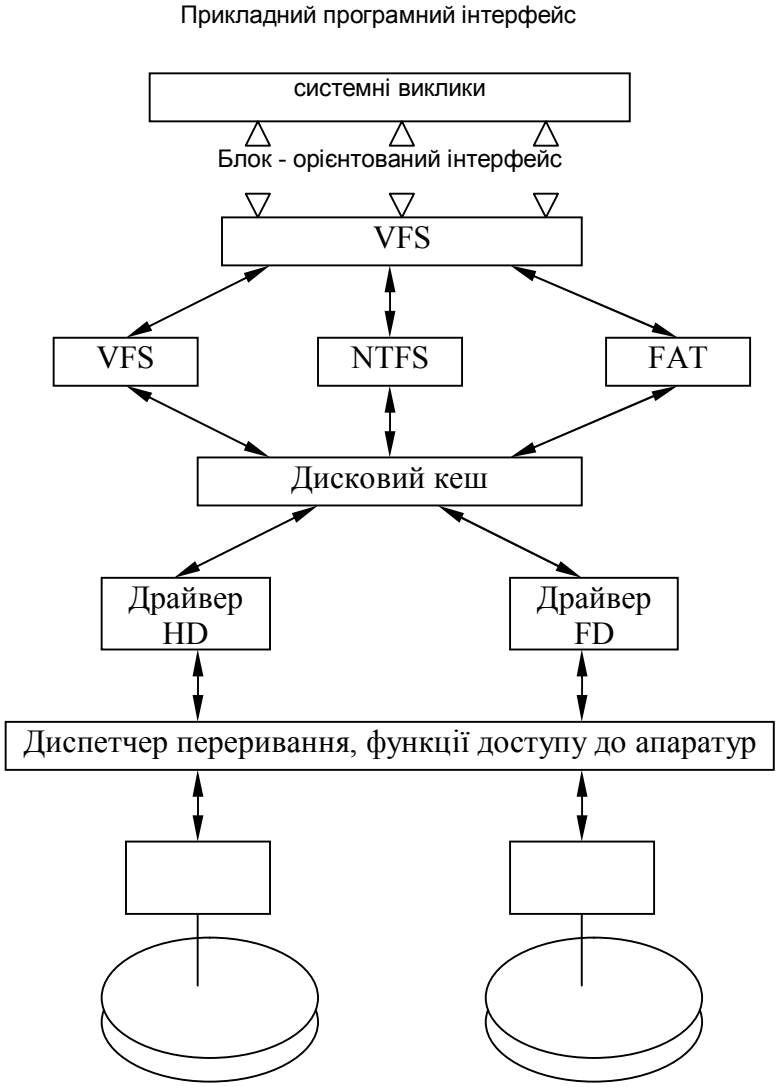


Рис. 9.1. Використання високорівневих драйверів

Ці абстракції створюються на більш високому рівні. Вони можуть підтримувати кілька файлових систем одночасно. Для цього в ОС установлюється декілька високорівневих драйверів (UFS, NTFS, FAT). Вони працюють із загальними апаратними драйверами, але по-своєму організують файлову систему для користувачів і прикладних процесів.

Для уніфікації подання різних файлових систем може використовуватися загальний драйвер верхнього рівня VFS (Virtual File System). Такий драйвер використовується, наприклад у системах UNIX.

В уніфікацію драйверів великий внесок внесла ОС UNIX. У ній всі драйвери розділені на 2 великих класи: блок-орієнтовані й байт-орієнтовані.

Наприклад, драйвери графічних і мережних пристроїв відносяться до байт-орієнтованих, а драйвери, керуючі пристроями прямого доступу, які зберігають інформацію в блоках фіксованого розміру, – до блок-орієнтованих (диск).

Можливість адресації блоків зумовлює для пристроїв прямого доступу можливість кешування даних в оперативній пам'яті. Це впливає на загальну організацію введення-виведення таких пристроїв.

9.11 Спеціальні файли

Спеціальними файлами називають іноді набори даних, які зберігаються на дисках. Вони використовуються для уніфікованого подання пристроїв введення-виведення.

Зі спеціальним файлом можна працювати так само, як і зі звичайним: відкривати, читати або записувати байти, а після завершення операції – закривати.

Для цього використовуються ті ж самі системні виклики, що й для роботи зі звичайними файлами: open, create, read, write, close.

Традиційно спеціальні файли містяться в каталозі /dev, хоча ніщо не заважає створити їх у будь-якому каталозі файлової системи. З появою

нового пристрою й, відповідно, нового драйвера адміністратор системи може створити новий запис (наприклад, за допомогою команди *mknod*).

9.12 Контрольні запитання до розділу 9

1. Яким чином узгоджуються швидкості обміну і кешування даних?
2. У яких режимах процесам надаються пристрої введення-виведення?
3. Яку модель інтерфейсу між пристроями введення-виведення і іншою частиною очислювальної системи підтримують сучасні ОС?
4. Чи підтримують сучасні ОС декілька файлових систем одночасно і яким чином?
5. Охарактеризуйте необхідність використання багаторівневих драйверів.
6. Які способи застосування високорівневих драйверів?

Розділ 10. Файлова система

10.1 Мета і завдання файлової системи

Файл – це іменована область зовнішньої пам'яті, в яку можна записувати й з якої можна зчитувати дані. Звичайно файли зберігаються в енергонезалежній пам'яті – диску. Однак є й виключення. Основні цілі використання файлу:

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

Файл може бути створений одним користувачем, а використовуватися іншим. При цьому можуть бути визначені права доступу до інформації.

Файлова система – частина ОС, що включає:

- сукупність всіх файлів на диску;
- набори структур даних для керування файлами: каталоги файлів, дескриптори файлів, таблиці розподілу вільного й зайнятого простору на диску;
- комплекс системних програмних засобів, що реалізують операції над файлами: створення, знищення, запис, читання, іменування й пошук файлів.

Файлова система дозволяє обходитися набором простих операцій над деяким абстрактним об'єктом, який названо **файлом**.

Файлова система екранує всі складності фізичної організації довгострокового зберігання даних і надає набір зручних у використанні команд для маніпулювання файлами.

Завдання, які вирішує ФС, залежать від способу організації обчислювального процесу в цілому. Найпростіший тип ФС – однокористувацька, однопрограмна. До їхнього числа належить MS-DOS.

ЇЇ функції наступні:

- іменування файлів;

- програмний інтерфейс для додатків;
- відображення логічної моделі файлової системи на фізичну організацію сховища даних;
- забезпечення стійкості файлової системи до збоїв живлення, помилок апаратних і програмних засобів.

Завдання ФС ускладнюються в однокористувальницьких мультипрограмних ОС. Прикладом такої ОС є OS/2. Тут додається нове завдання – спільний доступ до файлу з декількох процесів.

У цьому випадку, файл – поділюваний ресурс із усіма проблемами, що звідси з'являються.

У багатокористувацьких системах додається ще одне завдання – захист файлів від несанкціонованого доступу.

10.2 Логічна модель файлової системи

Логічна модель файлової системи матеріалізується у вигляді: дерева каталогів, що виводиться на екран, наприклад, за допомогою Norton Commander або Windows Explorer, у символьних складених іменах файлів і командах роботи з файлами.

Типи файлів

Звичайні файли містять інформацію довільного характеру. Більшість сучасних ОС (UNIX, Windows, OS/2) ніяк не обмежують і не контролюють вміст і структуру файлу.

Всі ОС повинні розпізнавати хоча б один тип файлів – їх власні виконувані файли.

Каталоги – особливий тип файлів. Вони містять системну довідкову інформацію про набір файлів, згрупованих користувачами за якоюсь неформальною ознакою (наприклад, один документ і т.п.).

У багатьох ОС у каталог можуть входити **спеціальні файли** – це фіктивні файли, що асоціюються із пристроями вводу–виводу. Вони використовуються з метою уніфікації механізму доступу до файлів і зовнішніх пристроїв. Спеціальні файли дозволяють користувачеві виконувати операції введення-виведення за допомогою звичайних команд запису у файл або читання з файлу. Ці команди обробляються спочатку програмами файлової системи, а потім на деякому етапі виконання запиту перетворюються ОС у команди керування відповідним пристроєм.

Ієрархічна структура файлової системи.

Більшість файлових систем має ієрархічну структуру, у якій рівні створюються за рахунок того, що каталог більш низького рівня може входити в каталог більш високого рівня (рис. 10.1).

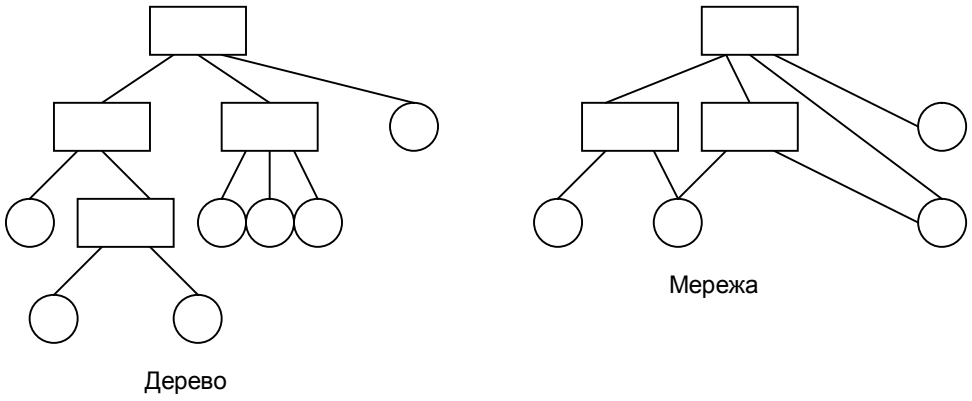


Рис. 10.1. Графи ієрархії каталогів

Граф, описуючий ієрархію каталогів, може бути деревом або мережею. Якщо файлу дозволено входити тільки в один каталог, файли утворять дерево. Якщо мережа – файл може входити відразу у декілька каталогів.

Наприклад, в MS DOS й Windows каталоги утворюють деревоподібну структуру, а в UNIX – мережну.

У деревоподібній структурі кожен **файл** є **листом.** каталог самого верхнього рівня називається **кореневим каталогом** або **коренем**.

Імена файлів

У файлових системах використовується три типи імен файлів: прості, складені й відносні.

Просте (коротке, символьне) ім'я ідентифікує файл у межах одного каталогу. Ці імена привласнюють користувачі з урахуванням обмежень ОС. Максимальна (у файловій системі FAT) довжина імені обмежується схемою 8.3 (ім'я – 8 символів, розширення – 3), а у файлових системах NTFS й FAT32, що входять до складу ОС Windows NT, ім'я файлу може містити до 255 символів.

В ієрархічних файлових системах різним файлам дозволено мати однакові прості символьні імена за умови, що вони належать різним каталогам.

Для однозначної ідентифікації в таких системах використовується так зване **повне ім'я.**

Повне ім'я являє собою ланцюжок, який є шляхом від кореня до даного файлу.

У деревоподібній файловій системі між файлом і його повним ім'ям є взаємно однозначна відповідність **один файл – одне повне ім'я**.

У випадку мережної структури має місце відповідність: **один файл – багато повних імен.**

Файл може бути також ідентифікований відносним ім'ям. Воно утвориться через поняття поточного каталогу. ОС фіксує ім'я поточного каталогу й використовує його як «добавку» до повного імені, використовуючи відносне ім'я. Наприклад, поточний каталог – USER, а відносне ім'я – main.exe. Повне ім'я – USER/main.exe.

Монтування

Обчислювальна система може мати кілька дискових пристроїв. Більш того, один фізичний пристрій може мати кілька логічних дисків.

Виникає проблема зберігання файлів у системі, що має кілька пристроїв зовнішньої пам'яті.

Перше рішення. На кожному із пристроїв розміщається автономна файлова система. Тобто є два незалежних дерева каталогів. Тут у повне ім'я файлу входить ідентифікатор відповідного логічного диску.

Друге рішення. Файлові системи поєднуються в єдину файлову систему, що описується єдиним деревом каталогів.

Така операція називають **монтуванням.**

При цьому ОС виділяє один дисковий пристрій, який називають **системним**. Нехай є дві файлові системи, розташовані на різних логічних дисках, причому один з них є системним. Файлова система, що розташована на системному диску, призначається кореневою. Для зв'язку ієрархій файлів у кореневій файловій системі вибирається деякий існуючий каталог. Після виконання монтування обраний каталог стає кореневим каталогом другої файлової системи. Через цей каталог файлова система, що монтують, приєднується як піддерево до загального дерева (рис. 10.2).

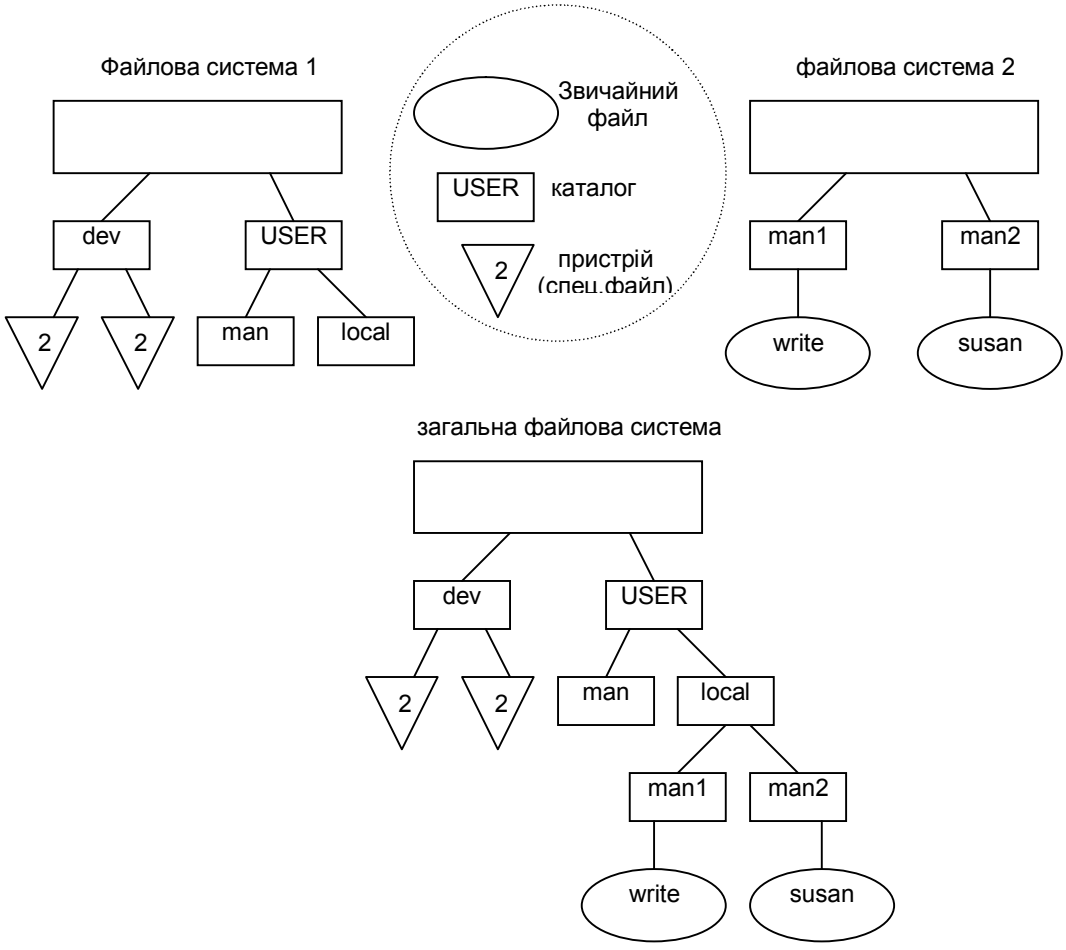


Рис. 10.2. Приклад монтування

Атрибути файлів

Атрибути – інформація, що описує властивості файлу. Можливі атрибути:

- тип файлу (звичайний файл, каталог, спеціальний файл);
- власник файлу;
- творець файлу;
- пароль для доступу до файлу;
- інформація про дозволені операції доступу до файлу;
- часи створення, останнього доступу й останньої зміни;
- поточний розмір файлу;
- максимальний розмір файлу;
- ознака «тільки для читання»;
- ознака «прихований файл»;
- ознака «системний файл»;
- ознака «архівний файл»;
- ознака «двійковий/символьний»;
- ознака «тимчасовий» (видалити по завершенні процесу);
- ознака блокування;
- ознака запису у файл;
- покажчик на ключове поле в записі;
- довжина ключа.

Конкретний перелік атрибутів визначається специфікою файлової системи. Наприлад, в MS-DOS зроблено як на рис.10.3.

<table>
<tr><td colspan="8">8</td><td colspan="3">3</td><td colspan="4">1</td><td colspan="4">4</td></tr>
<tr><td colspan="8">Ім'я файлу</td><td colspan="3">Розширення</td><td>R</td><td>A</td><td>H</td><td>S</td><td colspan="4">Резерви</td></tr>
<tr><td colspan="4">Резерви</td><td colspan="4">Час</td><td colspan="3">Дата</td><td colspan="4">Номер першого набору</td><td colspan="4">Розмір</td></tr>
</table>

Рис.10.3. Перелік атрибутів файлів

Іншим варіантом є розміщення атрибутів у спеціальних таблицях, коли в каталогах містяться тільки посилання на ці таблиці. У такій файловій системі (ОС UNIX) структура каталогу дуже проста (рис. 10.4).

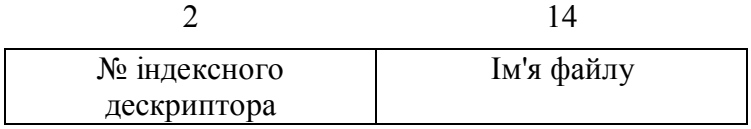


Рис. 10.4. Індексний дескриптор файлу

Індексний дескриптор файлу – таблиця, у якій зосереджені значення атрибутів файлу. Така система більш гнучка. Файл може бути включений відразу в кілька каталогів.

Логічна організація файлу

Ознаками, що відокремлюють один структурний елемент від іншого, можуть служити певні кодові послідовності або просто відомі програмі значення зсувів цих структурних елементів відносно початку файлу. Підтримка структури даних може бути або цілком покладена на програму, або, в тій або іншій мірі, цю роботу може брати на себе файлова система.

У першому випадку, файл представляється ФС як неструктурована послідовність даних. Програма формулює запити до файлової системи на введення-виведення, використовуючи загальні для всіх програм системні засоби, наприклад, указуючи зсув від початку файлу й кількість байт, які необхідно прочитати або записати. Потік байт, що надійшов до програми, інтерпретується відповідно до закладеної в програмі логіки.

Модель файлу, відповідно до якої його вміст представляється неструктурованою послідовністю (потоком) байт, широко використовується у більшості сучасних ОС (MS-DOS, Windows NT/2000). Неструктурована модель файлу дозволяє легко організувати поділ файлу між декількома

програмами: різні додатки можуть по-своєму структурувати й інтерпретувати дані, що містяться у файлі.

Інша модель файлу – **структурований файл**. Ця модель застосовувалася раніше (в OS/360). Підтримка структури файлу забезпечується файловою системою. Вона бачить файл як упорядковану послідовність **логічних записів**.

Програма може звертатися до ФС із запитами на введення-виведення на рівні записів. Наприклад, зчитавши запис 25 з файлу FILE.DOC, ФС має інформацію про структуру файлу, достатню для виділення будь-якого запису. ФС надає додатку доступ до запису, а вся подальша обробка даних, що міститься в цьому записі, виконується програмою.

Розвитком цього є системи, що підтримують складні структури даних і взаємозв'язок між ними.

Логічний запис –- найменший елемент даних, яким оперує програміст при обміні із зовнішнім пристроєм.

ФС може забезпечувати два способи доступу до логічних записів:

- послідовний доступ;
- прямий доступ (позиціонування на конкретний запис файлу).

10.3 Фізична організація файлової системи

Подання про файлову систему, як про ієрархічно організовану множину інформаційних об'єктів, має мало спільного з фактичним порядком зберігання файлів на диску.

Файл може бути розкиданий «шматочками» по всьому диску. Логічно об'єднані файли з одного каталогу зовсім не обов'язково є сусідніми на диску.

Принципи розміщення файлів, каталогів і системної інформації на реальному пристрої описуються **фізичною організацією файлової системи.**

Основним носієм є жорсткий диск, що складається з **пакету пластин.**

На кожній стороні пластини розміщені тонкі концентричні кільця – доріжки (tracks), на яких зберігаються дані, Кількість доріжок залежить від

типу диска. Нумерація доріжок починається з 0 від зовнішнього краю до центру. Коли диск обертається, головка зчитує дані з магнітної доріжки або записує їх на доріжку дискретними кроками. Кожен крок зсувається на 1 доріжку. У деяких дисках замість однієї головки є по головці на кожну доріжку.

Сукупність доріжок одного радіусу на всіх поверхнях всіх пластин називається **циліндром** (cylinder). Кожна доріжка розбивається на фрагменти, називані **секторами** (sectors) або **блоками** (blocks).

Всі доріжки мають однакове число секторів, у яких можна максимально записати таке ж саме число байт.

Сектор має фіксований для конкретної системи розмір, що виражається степенем двійки. Найчастіше – 512 байт. Оскільки доріжки різного радіусу мають однакове число секторів, щільність запису на доріжках різна – більша у центрі.

Сектор – найменша одиниця обміну даними з оперативною пам'яттю, яку можна адресувати.

Для того, щоб контролер міг знайти на диску потрібний сектор, необхідно задати йому всі складові адреси сектора: номер циліндра, номер доріжки й номер сектора.

Оскільки прикладній програмі потрібний не сектор, а деяка кількість байт, не обов'язково кратна розміру сектора, типовий запит включає читання декількох секторів, які містять необхідну інформацію з одного або двох секторів з надлишковими даними.

ОС при роботі з диском використовує власну одиницю дискового простору, що називають **кластером.** При створенні файлу на диску місце йому виділяють кластерами. Наприклад, розмір кластера може дорівнює 1024 байт.

Доріжки й сектори створюються в результаті виконання процедури фізичного (або низькорівневого) форматування. Форматування передує використанню диска.

Для визначення границь блоків на диск записується ідентифікаційна інформація.

Низькорівневий формат диску не залежить від типу ОС, що цей диск використовує.

Розмітку диску під конкретний тип файлової системи виконують процедури високорівневого або **логічного форматування**. При цьому визначається розмір кластера й записується інформація, необхідна для роботи файлової системи, у тому числі інформація про доступний і невикористовуваний простір, про границі областей, відведених під файли й каталоги, про ушкоджені області. Крім того, на диск записується **завантажник ОС** – програма (звичайно невелика), що починає процес ініціалізації ОС після включення живлення.

Перш, ніж форматувати диск під певну файлову систему, він може бути розбитий на розділи. **Розділ** – неперервна частина фізичного диску. ОС представляє його як **логічний пристрій** (логічний диск).

Оскільки файлова система, з якою працює одна ОС, не може, у загальному випадку, інтерпретуватися ОС іншого типу, логічні диски не можуть використовуватись ОС різного типу.

На кожному **логічному диску** може створюватися тільки **одна файлова система.**

На **різних логічних дисках** того самого фізичного диску можуть розташовуватися файлові системи **різного типу**.

Всі розділи одного диску мають однаковий розмір низькорівневого форматування. Але при високорівневому форматуванні в різних логічних дисках можуть бути встановлені файлові системи, розміри кластерів яких мають різні розміри.

ОС може підтримувати різні статуси логічних дисків, особливим чином їх розрізняючи. Одні розділи (диски) можна використати для завантаження модулів ОС, в інші можна встановлювати тільки програми й зберігати файли даних.

Один з розділів диска позначається як завантажувальний (або активний). Саме з нього зчитується завантажник ОС.

10.4 Фізична організація й адресація файлу

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

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

Неперервне розміщення – найпростіший варіант фізичної організації. При цьому файлу надається послідовність кластерів диску, що утворюють неперервну ділянку дискової пам'яті. **Основне достоїнство** – висока швидкість доступу, тому що витрати на пошук і зчитування кластерів файлу мінімальні. Мінімальний й обсяг адресної інформації – досить зберігати тільки номер першого кластера й обсяг файлу. Максимально можливий розмір файлу не обмежується.

Цей варіант має істотний недолік:

- неможливо передбачити, якого розміру повинна бути неперервна область, якщо файл при кожній модифікації може збільшувати свій розмір;
- велика проблема фрагментації (виникає багато вільних областей малого розміру);

Розміщення файлу у вигляді зв'язаного списку кластерів

При цьому способі на початку кожного кластера міститься покажчик на наступний кластер.

Адресна інформація мінімальна: розташування файлу задане одним числом - номером першого кластера. Фрагментація на рівні кластерів відсутня.

Недолік – складність реалізації доступу до довільно заданого місця файлу; потрібно простежити весь ланцюжок кластерів від початку.

Використання зв'язаного списку адрес

Файлу виділяється пам'ять у вигляді зв'язаного списку кластерів (рис. 10.5). Номер першого кластера запам'ятовується у записі каталогу, де зберігаються характеристики цього файлу. Інша адресна інформація відділена від кластерів файлу. З кожним кластером диска зв'язується деякий елемент – **індекс**. Таблиця індексів розташовується в окремій області диску і займає один кластер. Коли пам'ять вільна, всі індекси мають нульове значення.

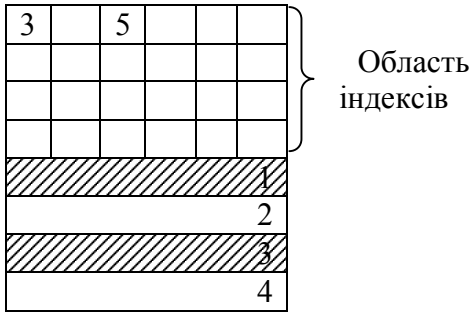


Рис. 10.5. Зв’язаний список адрес

Якщо деякий кластер N призначений деякому файлу, то індекс цього кластера стає рівним або номеру M наступного кластера даного файлу, або приймає спеціальне значення, що є ознакою того, що цей кластер є для файлу останнім. Індекс же попереднього кластера файлу приймає значення N, вказуючи на новопризначений кластер. Цей спосіб використовується у ФС FAT.

Ще один спосіб полягає у простому перерахуванні номерів кластерів, займаних файлом. Цей перелік і служить адресою файлу.

Недолік очевидний – довжина адреси залежить від розміру файлу й для великого файлу може скласти значну величину.

Достоїнство – висока швидкість доступу до довільного кластера, тому що тут використовується пряма адресація, а не перегляд ланцюжків.

Фрагментація також відсутня.

Як приклад розглянемо **фізичну організацію ФС FAT.**

10.5 Логічна організація FAT

Логічний розділ, відформатований під ФС FAT, складається з наступних областей (рис. 10.6):

- завантажувальний сектор. Містить програму початкового завантаження ОС. Вид цієї програми залежить від типу ОС, що буде завантажуватися із цього розділу;
- основна копія FAT. Містить інформацію про розміщення файлів і каталогів на диску;
- резервна копія FAT;
- область розміром у 32 рядки (16 Кбайт), що дозволяє зберігати 512 записів про файли й каталоги, тому що кожен запис каталогу складається з 32 байт;
- область даних. Призначена для розміщення всіх файлів і всіх каталогів, крім кореневого каталогу.

FAT підтримує всього два типи файлів: звичайний файл і каталог. ФС розподіляє пам'ять тільки з області даних.

Таблиця FAT (основна й копія) складається з масиву індексних покажчиків, кількість яких дорівнює числу кластерів області даних. Між кластерами й індексними покажчиками є взаємно однозначна відповідність.

Індексний покажчик може приймати наступні значення, що характеризують стани пов'язаного з ним кластера:

- кластер вільний;

- кластер використовується файлом і не є останнім кластером файлу. У цьому випадку, індексний покажчик містить номер наступного кластера файлу;
- останній кластер файлу;
- дефектний кластер;
- резервний кластер.

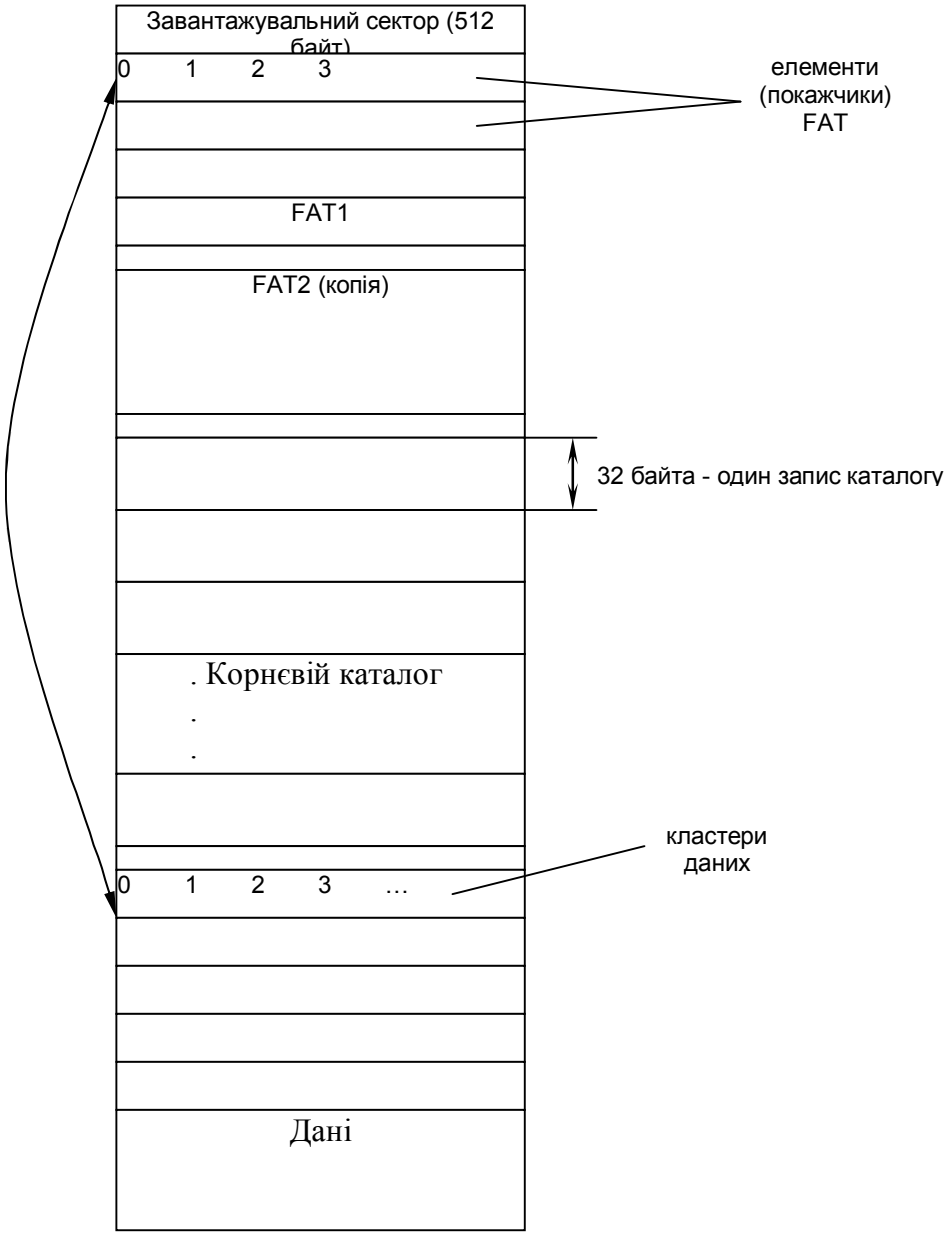


Рис. 10.6. Фізична організація FAT

Таблиця FAT є загальною для всіх файлів розділу.

У вихідному стані, після форматування, всі кластери вільні й всі індексні покажчики (крім тих, які відповідають резервним і дефектним блокам) приймають значення «кластер вільний».

При розміщенні файлу ОС переглядає FAT від початку, шукаючи перший вільний індексний покажчик. Після його виявлення в поле запису каталогу «номер першого кластера» фіксується номер цього покажчика. У кластер із цим номером записуються дані файлу. Він стає першим кластером файлу. Якщо файл уміщується в один кластер, то в покажчик відповідно записується значення «останній кластер файлу». Якщо розмір файлу більший, то ОС продовжує перегляд FAT і шукає наступний покажчик на вільний кластер. Після виявлення в попередній покажчик заноситься номер цього кластера, і він стає наступним кластером файлу.

Процес повторюється, поки не розміститься весь файл і не буде створений зв'язний список всіх кластерів файлу.

Розмір таблиці FAT і розрядність використовуваних у ній індексних покажчиків визначається кількістю кластерів в області даних. Для зменшення втрат через фрагментацію бажано робити кластери невеликими, а для скорочення обсягу адресної інформації й підвищення швидкості обміну – навпаки.

При формуванні диску під FAT звичайно приймають компромісне рішення, і розміри кластерів вибираються з діапазону від 1 до 128 секторів або від 512 байт до 64 Кбайт.

Існують кілька різновидів FAT індексних покажчиків, що відрізняються розрядністю – умовн позначають FAT12, FAT16 й FAT32. 12 розрядні покажчики підтримують 4096 кластерів, 16 розрядні – 65535, 32 розрядні – більш 4 млн.

При видаленні файлу із ФС у перший байт відповідного запису каталогу заноситься ознака того, що запис вільний, а в усі індексні покажчики файлу – нульові значення.

Інші дані запису каталогу, у тому числі номер першого кластера файлу, залишаються незмінними, що надає шанси для відновлення помилково вилученого файлу.

Використовуваний в FAT метод зберігання адресної інформації про файли не відрізняється великою надійністю – при розриві списку індексних покажчиків в одному місці (за різними причинами), втрачається інформація про всі наступні кластери файлу.

FAT12 й FAT16 застосовувалися в ОС MS-DOS, Windows 3.1, Windows NT/2000 й Windows 95/98. У зв'язку з ростом обсягів дисків вони витісняються FAT32.

Для поглиблення вивчення питань фізичної організації файлових систем, ознайомимось детальніше з ФС FAT32 та NTFS.

10.6 Файлова система FAT

Абревіатура FAT (File Allocation Table) або таблиця розміщення файлів відноситься до лінійної таблиці з відомостями про файли: їх найменуванням, атрибутами та іншими даними, що визначають місцезнаходження файлів у середовищі FAT.

Елемент FAT визначає фактичну область диску, у якій зберігається початок фізичного файлу.

У файловій системі FAT логічний дисковий простір поділяється на дві області: системну та область даних. Системна область логічного диску створюється та ініціалізується при форматуванні. Область даних логічного диску вміщує файли та каталоги, що підпорядковуються кореневому каталогу. На відміну від системної область даних, вона доступна через користувацький інтерфейс DOS. Системна область складається з наступних компонентів, які розміщені у логічному адресному просторі послідовно:

- Завантажувальний запис (або сектор);
- Зарезервовані сектори;
- Таблиця розміщення файлів;

- Кореневий каталог.

Таблиця розміщення файлів. Ця таблиця являє собою карту або образ області даних, у якій прописано стан кожної частки області даних.

Область даних розбивається на кластери. Кластер – це один або декілька суміжних секторів у логічному дисковому адресному просторі. У таблиці FAT кластери, що належать одному файлу (тобто некореневому каталогу), пов'язані у ланцюги. Для визначення номера кластера у системі FAT16 використовується 16 бітове слово, звідси можна мати 65536 кластерів. Файли або каталог займають ціле число кластерів. Останній кластер може бути задіяний не повністю, що призводить до значної втрати дискового простору при великому розмірі кластера. На дискетах кластер займає один або два сектори, а на жорстких дисках – у залежності від їх місткості. Таблиця розміщення файлів наведена на рис.10.6.

Ємність розділу	Кількість секторів у кластері	Розмір кластерів(Кбайт)
16-127	4	2
128-255	8	4
256-511	16	8
….	….	….
1024-2047	64	32

Рис.10.6. Таблиця розміщення файлів

Номер кластера завжди відноситься до області даних диску. Перший допустимий номер кластера завжди починається з 2. Номери кластерів відповідають області елементів таблиці розміщення файлів.

Логічний поділ області даних на кластери як сукупність секторів замість використання одиночних секторів має такий сенс: по перше, зменшується розмір самої таблиці FAT; зменшується можлива фрагментація файлів; прискорюється доступ до файлу, оскільки у декілька разів зменшується довжина ланцюжків фрагментів дискового проекту.

Однак великий розмір кластерів веде до неефективного використання області даних, особливо у випадку великої кількості маленьких файлів.

Наприклад, при розмірі кластера у 32 сектори, тобто розміром 16 Кбайт, середній розмір втрат становить 8 Кбайт. Якщо кількість файлів становить декілька тисяч, то витрати можуть становити більше ніж 100 Мбайт. Тому у більшості сучасних ОС розміри кластерів зменшуються до 512-4 Кбайт.

Ідея розміщення файлової системи з використанням таблиць може бути проілюстрована рис.10.7.

Рис. 10.7. Розміщення файлової системи з використанням таблиць

Файл починається з восьмого кластеру. Всього файл займає 11 кластерів, 18 кластер позначено спеціальним кодом F7 як дефектний, а 1C позначено кодом FF, що визначає кінець файлу. Вільні кластери позначено кодом 00.

Оскільки файли на диску змінюються, дані одного файлу можуть розміщуватись не у сусідніх кластерах, а утворювати досить складні ланцюги. Це призводить до суттєвого зменшення швидкодії у роботі з файлами.

У зв’язку з тим, що цілісність FAT дуже важлива, вона зберігається у двох ідентичних примірниках. Обновлюються копії одночасно. Для роботи використовується тільки перший екземпляр. Якщо з якихось причин він зруйнується, то буде використано другий екземпляр. Так у системі існує утиліта перевірки та відновлення файлової структури ScanDisc у ОС Windows, яка у разі виникнення невідповідності між першою та резервною копіями FAT пропонує відновити головну таблицю, використовуючи копію.

Кореневий каталог відрізняється від звичайного тим, що він має фіксоване число елементів. Для кожного файлу та каталогу зберігається інформація у відповідності з наступною структурою (рис. 10.8):

Розмір поля даних(байт)	Зміст поля даних
11	Ім’я файлу
1	Атрибути файлу
1	Резервне поле
3	Час створення
2	Дата створення
2	Дата останнього доступу
2	Зарезервовано
2	Час останньої модифікації
2	Дата останньої модифікації
2	Номер початкового кластера FAT
4	Розмір файла

Рис. 10.8. Структура корневого каталогу

Структура системи файлів – ієрархічна. У ній елементом каталогу може бути файл, який сам може бути каталогом (рис.10.9).

10.7 Файлові системи VFAT та FAT32

Раніше визначали, що дуже важливою характеристикою FAT було використання імен файлів формату “8.3”, де 8 символів відводять для вказівки імені файлу, а 3 – для його розширення. Пізніше до стандартної FAT16 додали ще 2 різновиди: VFAT(віртуальний FAT) та FAT32.

VFAT була розроблена для використання файлового введення-виведення у захищеному режимі. Також VFAT підтримує довгі найменування файлів. При цьому, вона зберігає сумісність із початковим варіантом FAT16, тобто підтримує формат “8.3”

Основні недоліки FAT та VFAT полягають у тому, що мають місце великі втрати на кластерізацію при великих розмірах логічного диску та обмеження розміру самого логічного диску. Це стало причиною розробки нової файлової системи з використанням старої ідеї використання таблиць FAT32.

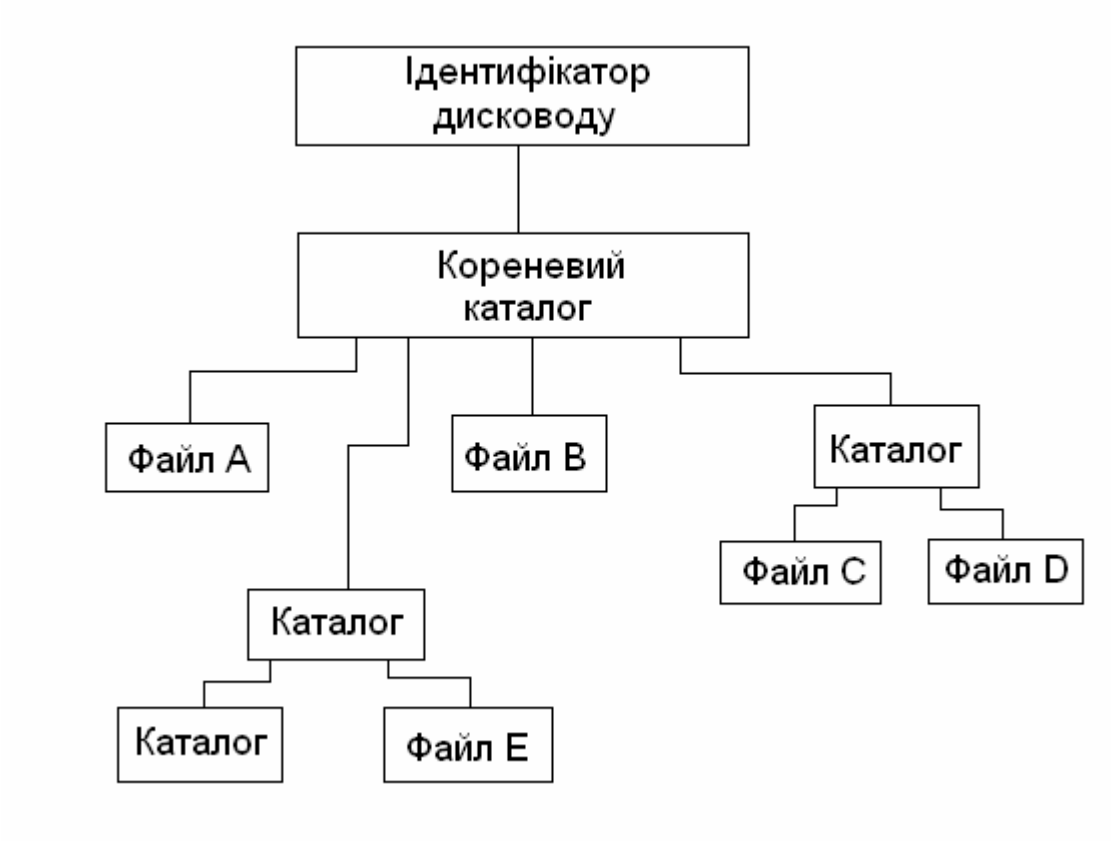


Рис. 10.9. Структура системи каталогів

FAT32 – 32 розрядна файлова система, яка доповнена деякими удосконаленнями у порівнянні з FAT16 та VFAT.

Принциповим є те, що FAT32 більш ефективно використовує дисковий простір. Вона використовує кластери локального розміру (оскільки у старій версії були обмеження на 65535 кластерів, і при збільшенні дискового простору треба було збільшувати розміри кластера). Наприклад, для дисків у 8 Гбайт FAT32 може використовувати 4 Кбайтові кластери. Як результат, економія у порівнянні з FAT16 досягла 10-15%.

FAT32 може переміщувати кореневий каталог та використовувати резервну копію замість стандартної. Кореневий каталог FAT32 задається у вигляді звичайного ланцюжка кластерів. Тобто кореневий каталог може знаходитись у довільному місці диску, що знімає старе обмеження на розмір кореневого каталогу у 512 елементів.

Окрім збільшення ємності FAT до 4 Тбайт (тера байт – 1024 Гбайт), FAT32 має необхідні вдосконалення у структурі кореневого каталогу. FAT16 вимагала щоб вся інформація кореневого каталогу була розміщена у одному дисковому кластері. При цьому вся його інформація мала не перевищувати 512 файлів. Необхідність представляти довгі імена та забезпечення сумісності з попередніми версіями FAT привели розробників до того, що для представлення довгого імені вони стали використовувати елементи каталогу. Структура елементів каталогів для FAT16 та FAT32 представлені на рис. 10.10, 10.11 та 10.12.

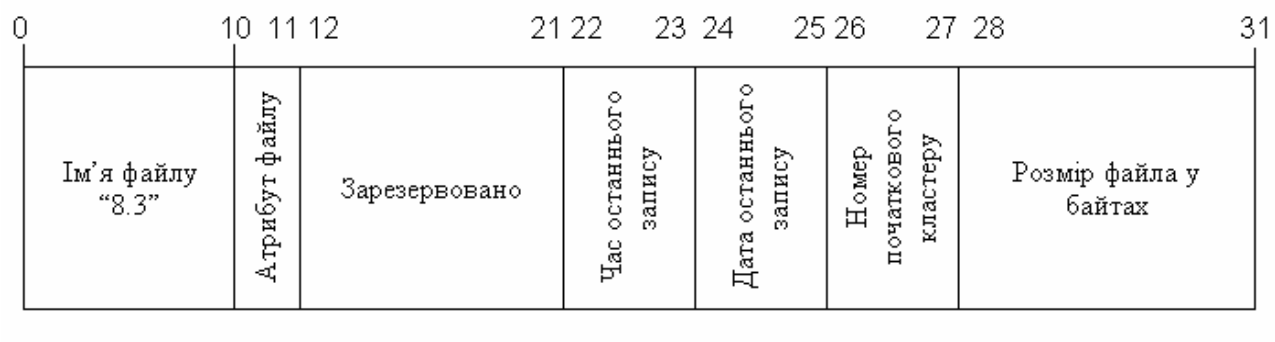


Рис 10.10. Елемент каталогу для короткого імені FAT16

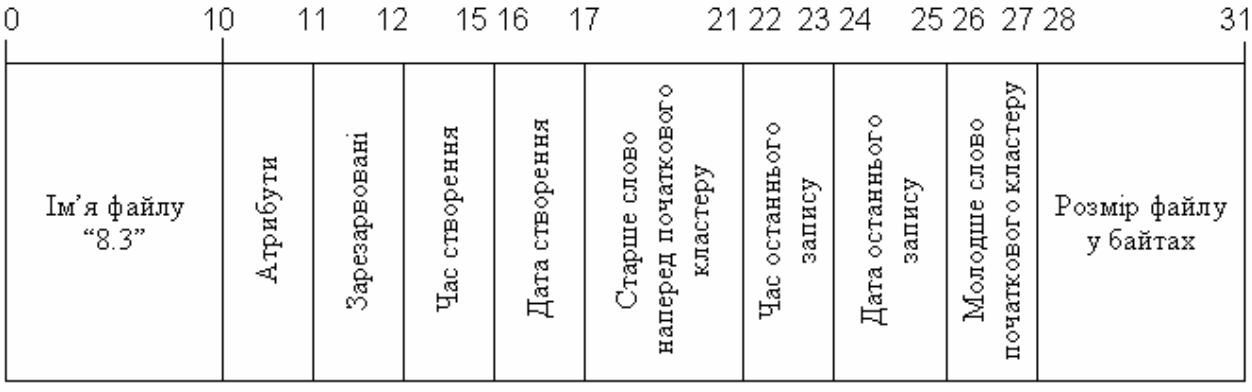


Рис. 10.11. Елемент каталогу для короткого імені FAT32

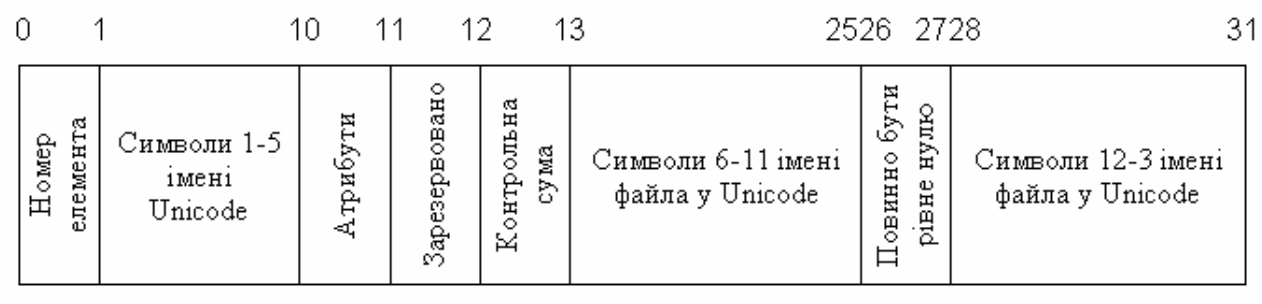


Рис. 10.12. Елемент каталогу для довгого імені FAT32

FAT32 успішно справляється з проблемою довгих імен у кореневому каталозі, але проблеми з обмеженням довжини повної файлової специфікації залишається. Тому рекомендовано обмежити довгі імена 75-80 символами, щоб залишити достатньо місця для задання шляху (180-185 символів).

10.8 Файлова система NTFS

Файли зберігаються у каталогах (здебільшого їх називають папками). На відміну від FAT, робота NTFS з дисками великого розміру виконується більш ефективно. У її розпорядженні також існують засоби для обмеження доступу до файлів та каталогів, введені механізми для підвищення надійності файлової системи, знято багато обмежень на максимальну кількість дискових секторів або кластерів.

NTFS добре працює при роботі з томами до 300-400Мбайт. Максимальний розмір тому та файлу становить 16 Ебайт (Екзабайт$\approx 2^{64} \approx$16000 млрд. Гігабайт). Кількість файлів у кореневому та кореневих каталогах не обмежена.

Основою структури каталогів NTFS є структура даних, яка дістала назву “бінарне дерево” (B-Tree). У файловій системі FAT каталог має лінійну структуру, яка спеціально не упорядкована, тому при пошуку файлу послідовно проглядаємо його із самого початку.

У випадку B-Tree (двійкове дерево, див. Н. Вірт “Алгоритми та структури даних”) структура каталогу являє собою збалансоване дерево з записами,

упорядкованими за алфавітом. Кожен запис, що входить до цього дерева, має атрибути файлу, показчик на відповідний файловий вузол, інформацію про час та дату створення, час і дату останнього оновлення та звертання, дані про розширені атрибути, лічильник звертань до файлу, довжину імені файлу, саме ім'я та іншу інформацію.

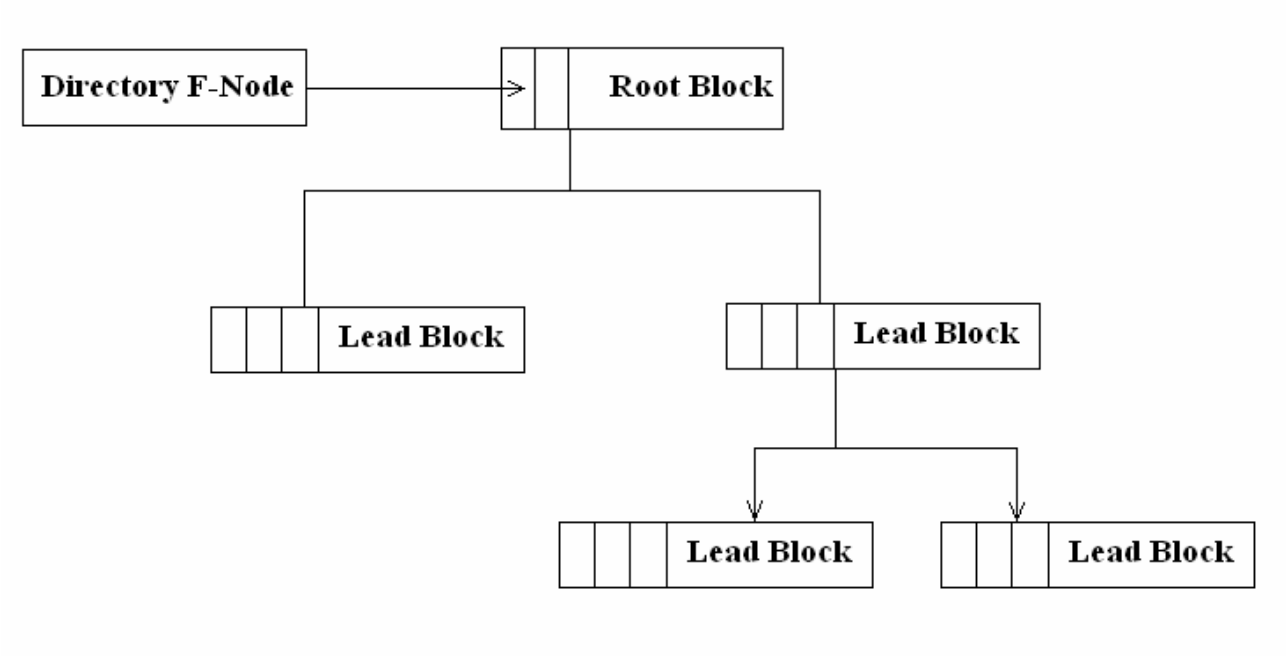


Рис. 10.13. Збалансоване двійкове дерево

При пошуку файлу у каталозі файлова система проглядає тільки необхідні гілки двійкового дерева. Такий метод більш ефективний ніж послідовне читання усіх записів у каталозі, що мало місце у FAT.

Для того щоб знайти необхідний файл у каталозі (точніше вказати на його інформаційну структуру F-Node), більшість записів взагалі читати не обов'язково. У результаті пошуку інформації про файл необхідно виконати значно меншу кількість операцій читання з диску.

Дійсно, якщо каталог містить 4096 файлів, при пошуку FAT потребує читання у середньому 64 секторів. NTFS буде читати лише 2-4 сектори для знаходження файлу.

Дійсно, при використанні 40 входів на блок (сектор диску) каталоги дерева з двома рівнями можуть мати 1640 входів, а каталоги дерева з трьома

рівнями – 65640 входів. Тобто, деякий файл може бути знайдено у типовому каталозі із 65640 файлів максимум за 3 звертання. У порівнянні з FAT потрібно буде, у найгіршому випадку, більш ніж 4000 секторів.

Одним з основних понять, що використовують при роботі з NTFS є поняття тому. NTFS поділяє дисковий простір тому на кластери – блоки даних, що адресуються як одиниці даних. Вона підтримує розміри кластерів від 512 байт до 64 Кбайт. Стандартним вважається кластер розміром 2 або 4 Кбайт.

Весь дисковий простір поділено на 2 нерівні частки (рис. 10.14).

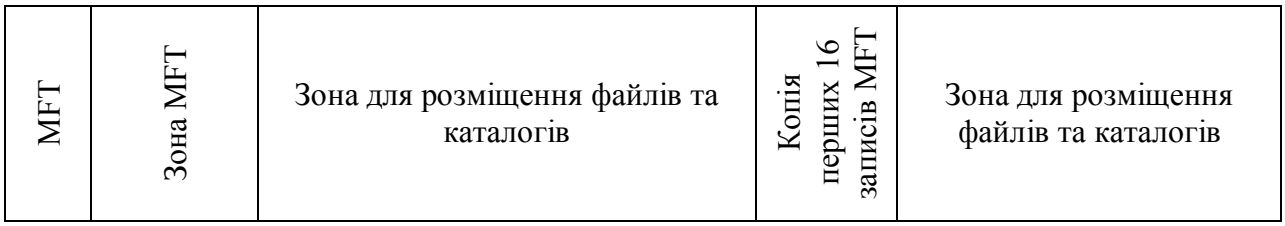


Рис. 10.14. Поділ дискового простору

Перші 12% диску відводяться під так звану MFT зону – простір, який може займати, збільшуючись у розмірі, головний службовий метафайл. MFT (Maste File Table) – спеціальний файл, головна системна структура даних, яка дозволяє визначати місцезнаходження всіх інших файлів.

MFT являє собою централізований каталог усіх інших файлів диску, у тому числі, і самого себе. MFT поділено на записи фіксованого розміру 1Кбайт, кожний запис відповідає конкретному файлу. Перші 16 файлів носять службовий характер і недоступні ОС, вони звуться метафайлами, причому самий першим з них є сам MFT.

Ці перші 16 елементів в MFT – єдина частина диску, що має строго фіксоване положення. Копія цих самих 16 записів зберігається у середині тому для надійності, оскільки вони дуже важливі. Інші частки MFT файлу можуть бути розміщені, як і інші файли, у довільних місцях диску –

поновити його положення можна за допомогою його самого, якщо “закріпитись” за основу – за перший елемент MFT.

Згадані 16 файлів NTFS (метафайли) носять службовий характер. Кожен з них відповідає за деякий аспект роботи системи. Метафайли знаходяться у кореневому каталозі NTFS тому. Усі вони починаються з символу імені “$”. Нижче наведено основні відомі метафайли та їх призначення:

$MFT – Can Master File Table.

$MFTmirr – Копія 16 записів MFT, розміщених у середині тому.

$LogFile – Файл підтримки операцій журналювання.

$Volume – Службова інформація (мітка тому, версія файлової системи)

$AttrDel – Список стандартних атрибутів файлів у томі.

$ – Кореневий каталог.

$Bitmap – Карта вільного місця тому.

$Boot – Завантажувальний сектор.

$Quota – Файл, де записані права користувачів на використання дискового простору.

$UpcaseФайл – Таблиця співвідношення заголовних та прописних букв у найменуваннях файлів. У NTFS найменування файлів записуються у Unicode (65 тис. різних символів), тому пошук великих та малих еквівалентів – складна задача.

Усі файли тому згадуються у MFT. У цій структурі зберігається вся інформація про файли, за винятком власне даних. Ім’я файлу, розмір, положення на диску окремих фрагментів – все це зберігається у відповідному записі. Якщо одного запису MFT замало, то використовується декілька записів, причому не обов’язково послідовних.

Якщо файли мають дуже маленький розмір, то дані файлу зберігаються безпосередньо в MFT у межах одного запису.

Файл у томі з NTFS ідентифікується з так званою файловим посиланням, (File Reference), яке визначається як 64-розрядне число. Воно складається з номеру файлу, що відповідає позиції його файлового запису у MTF, та

номеру послідовності. Цей номер збільшується кожний раз, коли ця позиція у MTF використовуються повторно, що дозволяє файловій системі NTFS виконувати внутрішні перевірки цілостності.

Кожен файл задається за допомогою потоків (streams), тобто у нього немає “просто даних”, а є “потоки”.

Щоб визначитись з потоками, достатньо вказати, що один з потоків несе звичайне поняття – дані файлу. Але більшість атрибутів файлу – це теж потоки.

Таким чином визначається, що базова сутність у файлу тільки одна – номер у MTF, а все інше, включаючи потоки – опції.

Даний підхід може бути ефективно використано, наприклад, файлу можна додавати потоки та записувати у них будь-які дані.

Структурні атрибути файлів такі:

- стандартна інформація про файл. Традиційні атрибути Read Only, Hidden, Archive, System, час створення, останньої модифікації, число каталогів з посиланням на файл;

- список атрибутів. Список атрибутів, з яких складається файл, файлова вказівка на файловий запис в MTF, де розміщено кожний з атрибутів. Останній використовується, якщо файлу необхідно більш ніж один запис у MTF;

- ім’я файлу. У символах Unicode файл може мати декілька атрибутів – імен файлу;

- дескриптор захисту. Хто має доступ до файлу, його власник та захист від несанкціонованого доступу;

- дані. Власні дані файлу, його склад. Файл може мати додаткові атрибути;

- корінь індексу, розміщення. Атрибути, що використовуються для індексу, бітова картка індексів імен файлів у великих каталогах (тільки для каталогів);

- розширені атрибути NTFS.

Ім’я файлу на відміну від FAT може використовувати будь-які символи, вказуючи повний перелік національних атрибутів, оскільки символи представлені 16 бітами, тобто є 65535 різних символів. Максимальний розмір імені файлу 255 символів.

Каталог – це спеціальний файл, у якому є вказівки на інші файли та каталоги, створюючи ієрархічну будову даних на диску. Файл каталогу поділено на блоки, кожний з яких має у своєму складі ім’я файлу, базові атрибути та вказівку на елемент MTF, який представляє повну інформацію про елемент каталогу.

Головний каталог диску – кореневий – нічим не відрізняється від звичайних каталогів, окрім спеціальної вказівки на нього з початку метафайла MTF.

Внутрішня структура каталогу являє собою бінарне дерево. Це дерево розміщує імена файлів таким чином, щоб пошук файлу виконувався за допомогою отримання двозначних відповідей на запити про положення файлу. Бінарне дерево може дати відповідь на запитання: у якій групі, відносно даного елементу, знаходиться дане ім’я – вище чи нижче. Починаємо з такого запиту до середнього елементу. Кожна відповідь звужує зону пошуку у 2 рази. Якщо файли відсортовані за алфавітом, то відповідь отримаємо порівнянням початкових букв.

Область пошуку, зменшена попереднього у 2 рази, знову починається з середини і т.д.

10.9 Основні відмінності FAT та NTFS

Накладні витрати на зберігання службової інформації FAT різняться від NTFS більшою компактністю та меншою складністю. У більшості томів FAT для зберігання таблиці розміщення, у якій знаходиться інформація про всі файли тому, використовується не більше 1 Мбайт. Це дозволяє формувати FAT не тільки на жорсткі диски, але й на флоппі-диски.

У NTFS службові дані займають більше місця. Кожний елемент каталогу займає 2 Кбайти. Однак це має і свої переваги. Так вже згадувалось, що вміст файлів об’ємом до 1500 байт може зберігатися безпосередньо у каталозі.

Системою NTFS не рекомендовано користуватися для форматування розділів об’ємом менше 50 Мбайт. Відповідно великі накладні витрати призводять до того, що для малих розділів службові дані можуть займати до 25% загального об’єму даних.

Корпорація Microsoft рекомендує використовувати FAT для розділів об’ємом до 256 Мбайт, а NTFS – для розділів від 400 Мбайт та більше.

Наступний критерій для порівняння – розмір файлів. Розділи FAT мають об’єм до 2 Гбайт, FAT32 – до 4 Тбайт, однак завдяки своїй внутрішній побудові, розділи FAT краще всього працюють при розмірах 200 Мбайт та менше.

Розділи NTFS можуть досягати 16 Эбайт, хоча існують апаратні проблеми, завдяки яким розміри файлів зменшуються до 2 Тбайт.

Розміри FAT можуть бути використані практично з усіма операційними системами. З розділами NTFS може працювати лише ОС Windows NT, хоча існує утиліта NTFS DOS, що дозволяє читати дані NTFS на комп’ютері, що моделює режим DOS.

Щодо безпеки. Розділи FAT не забезпечують максимальної безпеки. З іншої сторони, розділи NTFS забезпечують максимальну безпеку як файлів, так і каталогів. Для розділів FAT можна встановити загальні правила доступу до каталогів мережі. Однак такий захист не завадить користувачу з локальним входом отримати доступ до файлів. NTFS більш захищена. Розділи NTFS можуть заборонити чи обмежити доступ як до віддалених, так і до локальних користувачів. Тобто до захищених файлів зможуть звернутися лише користувачі, які мають відповідні права.

Windows NT має у своєму розпорядженні спеціальну утиліту CONVERT.EXE, що дозволяє перетворити томи FAT у еквівалентні томи NTFS, однак зворотне перетворення неможливе.

У останній час, завдяки прогресу у конструкції дискових механізмів, значно перевищені можливості FAT у 8.4 Гбайта. Тому при роботі у середовищі Windows часто використовують FAT32 або NTFS.

У файловій системі NTFS використаний метод збалансованих бінарних дерев для зберігання та пошуку інформації про місцезнаходження файлу.

Структура каталогу являє собою збалансоване дерево із записами, що впорядковані за алфавітом. Кожний запис, що входить у склад B-Tree дерева містить атрибути файлу, вказівник та відповідний файловий вузол, інформацію про час та дату створення файлу, час та дату останнього оновлення та звертання, розмір даних, лічильник звертань до файлу, розмір імені файлу, саме ім’я та деяку іншу інформацію.

NTFS при пошуку файлу у каталозі проглядає тільки необхідні гілки двійкового дерева. Цей метод набагато ефективніший ніж послідовне читання усіх записів у каталозі, що має місце у FAT.

Для того, щоб знайти деякий файл у каталозі (точніше вказівник на його інформаційну структуру), більшість записів вказувати не треба. Тобто використана впорядкованість для суттєвого зменшення кількості операцій читання диску.

Збалансованість дерева зменшує глибину дерева і, як наслідок, теж зменшує тривалість пошуку.

10.10 Файлові операції

Файлова система надає користувачам певний набір операцій для роботи з файлами.

Але які б операції не здійснювалися, ОС необхідно виконати ряд операцій, універсальних для всіх виконуваних дій:

- за символьним іменем файлу знайти його характеристики, що зберігаються у файловій системі;
- скопіювати характеристики файлу в оперативну пам'ять;

- на підставі характеристик файлу перевірити права користувача на виконання запитаної операції (читання, запис, видалення й т.п.);
- очистити область пам'яті, відведену під тимчасове зберігання характеристик файлу.

ОС може виконувати послідовність дій над файлом двома способами:

- для кожної операції виконуються як універсальні, так й унікальні дії. Така схема іноді називається схемою без запам'ятовування стану операції;
- всі універсальні дії виконуються на початку й наприкінці послідовності операцій, а для кожної проміжної операції виконуються тільки унікальні дії.

Більшість ФС підтримують другий спосіб організації файлових операцій, як більш економічний і швидкий.

Проте, перший спосіб є більш стійким до збоїв у роботі системи, тому що кожна операція є самодостатньою й не залежить від результату попередньої.

Відкриття файлу

Системний виклик **open** працює із двома аргументами: символьним ім'ям файлу, що відкривається, і режимом відкриття файлу. Режим відкриття говорить системі, які операції будуть виконуватися над файлом у процесі роботи до закриття файлу за системним викликом **close**. Наприклад, тільки читання, тільки запис або читання й запис.

Обмін даними з файлом

Для обміну даними з файлом (попередньо відкритим) використовуються процедури read і write. У тому випадку, коли необхідно явно вказати, з якого байта файлу необхідно читати або записувати дані, використовуються додаткові системні виклики.

На основі системних викликів введення-виведення будуються могутніші бібліотечні функції введення-виведення, що складають прикладний інтерфейс ОС.

Блокування файлів

Блокування файлів й окремих записів у файлах є засобом синхронізації між працюючими в кооперації процесами, які намагаються використати один і той же файл одночасно.

Багатокористувацькі ОС звичайно підтримують спеціальний системний виклик, що дозволяє програмістові встановити й перевірити блокування на файл і його окремі області.

Процеси, що кооперуються, обов'язково повинні перевіряти наявність блокування на файл, щоб синхронізувати свою роботу.

Контроль доступу до файлів

Файли – окремий вид поділюваних ресурсів, доступ до яких ОС повинна контролювати.

У кожного об'єкта доступу існує власник. Власником може бути як окремий користувач, так і група користувачів. Власник об'єкта має право виконувати з ним будь-які припустимі для даного об'єкта операції. У багатьох ОС існує особливий користувач (superuser, root, administrator), що має усі права стосовно будь-яких об'єктів системи, не обов'язково будучи їхнім власником. Під таким ім'ям працює адміністратор системи, якому необхідний повний доступ до всіх файлів і пристроїв для керування політикою доступу.

Розрізняють два основних підходи до визначення прав доступу:

Вибірний доступ. Має місце, коли для кожного об'єкта сам власник може визначити припустимі операції з об'єктами. Цей підхід називається так само довільним доступом, тому що дозволяє адміністраторові й власникам об'єктів визначити права доступу довільним образом, за їхнім бажанням. Між

користувачами й групами користувачів у системах з вибірним доступом немає жорстких ієрархічних взаємин, які визначені за замовчуванням, і тільки адміністратор, за замовчуванням, наділяється всіма правами.

Мандатний доступ. Система наділяє користувача певними правами стосовно кожного поділюваного ресурсу, залежно від того, до якої групи користувача віднесено.

Від імені системи виступає адміністратор, а власники об'єктів не мають змоги управляти доступом до них за своїм розсудом.

Всі групи користувачів у такій системі утворять чітку ієрархію, причому кожна група користується всіма правами доступу групи більш низького рівня ієрархії.

Членам якої-небудь групи не дозволяється надавати свої права членам груп більш низьких рівнів ієрархії.

Мандатні системи доступу вважаються більш надійними, але менш гнучкими, звичайно вони приймаються в спеціалізованих обчислювальних системах з підвищеними вимогами до захисту інформації.

Механізм контролю доступу

Кожен користувач і кожна група користувачів звичайно має символьне ім'я, а також унікальний числовий ідентифікатор.

При виконанні процедури логічного входу в систему користувач повідомляє своє символьне ім'я й пароль, а ОС визначає відповідні числові ідентифікатори користувача й груп, у які він входить.

Всі ідентифікаційні дані, у тому числі імена й ідентифікатори користувачів і груп, паролі користувачів, а також відомості про входження користувача в групи зберігаються

Вхід користувача в систему породжує процес - оболонку, що підтримує діалог з користувачем і запускає для нього інші процеси.

Процес - оболонка одержує від користувача символьне ім'я й пароль і знаходить по них числові ідентифікатори користувача і його груп. Ці

ідентифікатори зв'язуються з кожним процесом, запущеним оболонкою для даного користувача. Говорять, що процес виступає від імені даних користувача й даних груп користувачів.

Визначити права доступу до ресурсу – означає визначити для кожного користувача набір операцій, які йому дозволено застосовувати до даного ресурсу.

У різних ОС для тих самих типів ресурсів може бути визначений свій список диференційованих операцій доступу.

Для файлових об'єктів цей список містить наступні операції:

- створення файлу;
- знищення файлу;
- відкриття файлу;
- закриття файлу;
- запис у файл;
- доповнення файлу;
- пошук у файлі;
- одержання атрибутів файлу;
- установка нових значень атрибутів;
- переписування;
- виконання файлу;
- читання каталогу;
- зміна власника;
- зміна прав доступу.

У найбільш загальному випадку, права доступу можуть бути описані матрицею прав доступу, де стовпці відповідають всім файлам системи, рядки – всім користувачам, а на перетині рядків і стовпців вказуються різні операції (рис. 10.15). Практично у всіх ОП матриця прав доступу зберігається "вроздріб", тобто для кожного файлу й каталогу створюється так званий **список керування** доступом, у якому описуються права користувачів і групи користувачів на виконання операцій щодо цього файлу й каталогу. Список

керування доступом є частиною характеристик файлу або каталогу й зберігається на диску у відповідній області.

		modern.txt	win.exe	…	…
Імена користувачів	Kira	читати	виконувати		
	Victor	читати	виконувати		
	Nataly	-	-	створити	
	. . .				

Рис. 10.15. Механізм прав доступу

Однак, не всі файлові системи підтримують список керування доступом, наприклад, його не підтримує ФС FAT, оскільки вона розроблялася для однокористувацької MS DOS.

Список керування доступом з доданим до нього ідентифікатором власника називається **характеристиками обов'язку.**

10.11 Контрольні запитання до розділу 10

1. Що означає поняття файлу?
2. Яка мета використання файлів?
3. Які складові файлової системи?
4. Які бувають типи файлів?
5. Дайте визначення поняттю **монтування.**
6. Перерахуйте можливі атрибути файлів.
7. Дайте визначення логічній організації файлової системи.
8. Дайте визначення фізичній організації файлової системи.
9. Порівняйте логічну і фізичну організацію файлу.
10. Охарактеризуйте логічну і фізичну організацію файлової системи.

11.Наведіть логічну і фізичну організацію файлової системи FAT.

12.Які відмінності між файловими системами FAT16 та FAT32?

13. Охарактеризуйте файлову систему NTFS.

14. Порівняйте архітектури FAT та NTFS.

15. Перерахуйте типи файлових операцій.

16. Дайте визначення поняттю мандатного доступу.

Розділ 11. Архітектурні особливості побудови ОС

11.1 Особливості побудови ОС UNIX

Вище вивчались складові операційних систем, а на закінчення курсу хотілось би познайомитись з основними архітектурними особливостями побудови ОС.

У цьому сенсі, найбільший інтерес представляє ОС Unix. Це мультипрограмна та багатокористувацька ОС. У свій час вона проектувалась, як засіб для розробки програмного забезпечення. Унікальність її полягає в тому, що вона була розроблена всього двома розробниками (Кен Томпсон та Денніс Рітчі). Вони націлювали її для використання на міні-ЕОМ, і тому вона орієнтувалась на досить скромні обчислювальні ресурси. Вона має досить ефективну командну мову та незалежну від пристроїв файлову систему.

Вся ОС була написана мовою програмування високого рівня (мова С), а тому вона легко переноситься.

Дуже важливою особливістю ОС є те, що користувачі мають можливість направляти виходи однієї програми безпосередньо на вхід другої шляхом реалізації так званих каналів (pipe). Як результат, великі програмні комплекси можна створювати шляхом композиції існуючих програм, а не шляхом розробки нових. Це поклало початок новому напрямку в програмуванні – створенню технологій програмування подібно маршрутних технологічних карт для виготовлення механічних деталей.

Власне ОС Unix поставляється з великим набором системних та прикладних прграм, включаючи редактори текстів, програмовані інтерпретатори командної мови, компілятори С, С++, асемблера та багато інших мов.

Зупинимось на наборі основних понять, щоб зрозуміти основні відмінності Unix від звичного для багатьох ОС Windows.

Віртуальна машина. ОС – багатокористувацька. Кожному користувачеві надається віртуальний комп'ютер, у якому є всі необхідні ресурси. Процесор розподіляє час на основі циклічної дисципліни обслуговування (або так званої карусельної диспетчеризації з використанням динамічних приорітетів, щоб забезпечити рівні права в обслуговуванні).

Поточний стан такого віртуального комп'ютера називають образом. Власне процес у нашій термінології – це виконання образу.

Такий образ має наступні складові:

- образ пам'яті;
- стан загальних регістрів процесора;
- стан відкритих файлів;
- поточна директорія, каталог файлів та іншу інформацію.

Образ процесу під час виконання розміщується в основній пам'яті. Як правило, підтримується сторінковий механізм організації віртуальної пам'яті.

Власне образ пам'яті поділяється на три логічні сегменти:

- сегмент реєнтабельних процедур (розміщується, починаючи з нульової адреси у віртуальному адресному просторі процеса);
- сегмент даних (розташований безпосередньо за сегментом процедур і може зростати у бік більших адрес);
- сегмент стеку (використовується при викликах підпрограм та при перериваннях).

Користувач

Unix орієнтована на мультитермінальну роботу. Для початку роботи людина-користувач має увійти в систему, ввести з вільного терміналу своє ім'я, а можливо і пароль. Кожен користувач має реєстраційний запис і зветься зареєстрованим користувачем системи. Реєстрацію здійснює адміністратор системи. Користувач не може змінити своє ім'я, але може змінити свій пароль.

Всі користувачі працюють з файлами. Файлова система має деревоподібну організацію. Проміжні вузли є каталогами, що мають

посилання на інші каталоги, листя дерева відповідають листям або порожнім каталогам.

Кожному зареєстрованому користувачу відповідає свій каталог, який зветься «домашнім» (home) каталогом користувача. Користувач має необмежений доступ до свого домашнього каталогу, до всіх каталогів і файлів. Він може модифікувати та створювати нові файли. Потенційно він може мати доступ і до інших файлів, але це залежить від прав доступу.

Інтерфейс користувача

Традиційний спосіб взаємодії між користувачем та ОС базується на використанні командних мов, а також має місце використання графічного інтерфейсу типу Windows. Після входження користувача в систему запускається один із командних інтерпретаторів, яких може бути декілька. Загальна назва для такого інтерпретатора – оболонка (shell), оскільки вона являє собою зовнішнє середовище ядра ОС.

Командний інтерпретатор видає запрошення на ввід командного рядку, що являє собою просту команду, конвеєр команд або послідовність команд. Після виконання командного рядку на екран терміналу або у відповідний файл оболонка (shell) знову видає запрошення доти, доки користувач не вирішить вийти з системи.

Одним з різновидів управління є вказівка командного файлу, який містить такі ж командні рядки.

Привілейований користувач

Ядро ОС ідентифікує користувача за його ідентифікатором (user identifier), яким позначають користувача. Крім того, кожний користувач відноситься до деякої окремої групи (group identifier). Ці ідентифікатори для кожного зареєстрованого користувача зберігаються у спеціальних файлах системи і приписані процесу, у якому виконується командний інтерпретатор.

Ці ідентифікатори наслідуються кожним новим процесом, що запущено від імені конкретного користувача.

Адміністратор має значно більші можливості, ніж звичайні користувачі. Такий користувач має назву (SuperUser) суперкористувач. Він має право контролю над системою.

Для звичайних користувачів встановлені обмеження: максимальний розмір файлу, максимальна кількість сегментів поділюваної пам'яті, розмір віртуального процесору і т.п. Суперкористувач має право змінити обмеження для інших користувачів.

Команди та командний інтерпретатор

Оболонка (shell) – це механізм взаємодії між користувачем та системою. Інтерпретатор аналізує рядки та виконує відповідні функції. Склад командного рядка наступний: команда (перелік аргументів, поділених пропусками), оболонка (поділяє командний рядок на компоненти). Відповідний файл завантажується, і йому надається доступ до вказаних у команді аргументам.

Кожен з командних мов shell складається з трьох частин:

- службові інструкції, що дозволяють маніпулювати із рядками тексту та будувати складні команди на базі простих;
- вбудовані команди, що виконуються безпосередньо інтерпретатором;
- команд, що зображуються окремо виконуваними файлами.

Команди останньої категорії складаються із стандартних команд (системні утиліти) та команд, що створені користувачами. Для того щоб створений користувачем файл можна було запускати як команду оболонки (shell), достатньо визначити в одному із вихідних файлів функцію з іменем main, яке у свою чергу має бути глобальним, тобто перед ним не має бути вказано слово static. Командний інтерпретатор створить новий процес, запустить на виконання програму, починаючи зі слова main. Тіло (або вміст)

функції main може бути довільним, але потрібно виконувати деякі правила. Наприклад, можна використовувати тексти мовою С.

Процеси

Під цим терміном розуміється процес у класичному його розумінні, тобто як програма, що виконується у власному віртуальному 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 та запускає відповідну програму, починаючи зі слова main. Ця програма, у свою чергу, може створити процес та запустити в ньому іншу програму.

Для створення нового процесу та запуску в ньому програми використовуються два системні виклики fork та exec (<ім'я файлу>). Системний виклик fork створює адресний простір процесу. Для дочірнього процесу створюються копії всіх сегментів даних. Кожен процес має своїх батьків, але в свою чергу може бути батьком багатьох процесів. Початковий (нульовий процес) є особливим, він створюється в результаті завантаження системи. Тобто ця ОС підтримує структурований спосіб організації процесів у вигляді дерева.

Виконання процесів

Процес може виконуватись в одному із двох станів: користувацькому та системному. У користувацькому стані процес виконує користувацьку програму та має доступ до користувацького сегменту даних. У системному стані процес виконує програми ядра та має доступ до системного сегменту даних.

Коли користувацькому процесу треба виконати системну функцію, він виконує системний виклик. З моменту появи системного виклику процес вважається системним, оскільки фактично виконується виклик ядра системи,

як підпрограми. Тобто користувацький та системний процеси являють собою дві фази одного й того ж процесу, але вони ніколи не перехрещуються між собою. Кожна фаза використовує свій власний стек. У цих ОС виконується розподіл часу, тобто кожному процесу надається квант часу. Процес або сам закінчується до завершення відведеного йому кванту, або він відкладається до завершення кванту.

Механізм диспетчеризації характеризується таким розподілом часу, щоб процеси закінчувались у порядку їх виникнення. Користувацьким процесам приписують приорітети в залежності від кількості процесорного часу, що вони отримують. Процесам, що отримали багато процесорного часу, визначають більш низькі приорітети; у той же час процесам, які отримали невелику кількість часу процесора, навпаки, підвищують приорітет.

Підсистема введення-виведення

Функції цієї підсистеми задаються системними викликами: open, close, read, write, seek.

При читанні можливі три операції, у кожній з яких читання виконується послідовно:

- якщо це перше читання з файлу, то воно виконується послідовно з самого початку файлу;
- якщо операції читання передувала інша операція читання з цього файлу, то виконувана операція надасть дані, що безпосередньо наступними за попередніми;
- якщо читанню передувала операція seek, то читання виконується послідовно від точки зміщення, що вказана у команді seek.

Звертаємо увагу на те, що всі ці виклики відносяться до послідовного доступу, а ефект прямої адресації досягається за допомогою команди seek, що зміщує поточну позицію файлу.

Існує ще декілька примітивів, що дозволяють отримувати та задавати інформацію про файли та термінали. Фізичні пристрої представлені спеціальними файлами у єдиній структурі файлової системи.

Перенаправлення введення-виведення

Механізм перенаправлення введення-виведення є одним з найбільш простих механізмів.

Unix – інтерактивна система. Історично склалось так, що програми звичайно вводили текстові рядки з терміналу та виводили результати на екран терміналу. Для того, щоб забезпечити більш гнучкий спосіб використання таких програм, треба було зуміти забезпечити їм вивід інформації з файлів інших програм і направити її у файл або ввод-вивід конкретної програми .

Реалізація цього механізму базується на наступних властивостях ОС. По перше, будь-який введення-виведення трактується як ввід з деякого файлу і вивід у деякий файл. Клавіатура та екран терміналу теж інтерпретуються як файли. По-друге, доступ до будь-якого файлу виконується через його дескриптор. Фіксуються три значення дескрипторів файлу. Файл з дескриптором 1 зветься файлом стандартного вводу (stdin), файл з дескриптором 2 – файлом стандартного виводу (stdout) та файл з дескриптором 3 – файлом стандартного виводу діагностичних повідомлень (stder). По-третє, програма, запущена в деякому процесі «наслідує» від породжуваного процесу всі дескриптори відкритих файлів.

У головному процесі інтерпретатора командної мови файлом стандартного вводу є клавіатура терміналу користувача, а файлом стандартного виводу та виводу діагностичних повідомлень – екран терміналу. Однак, при запуску будь-якої команди можна повідомити інтерпретатору (засобами командної мови), який саме файл або вивід якої програми має слугувати файлом стандартного вводу для запуску якої програми, і який файл або вивід якої програми має слугувати файлом

стандартного виводу або виводу діагностичних повідомлень для програми, яка запускається.

Інтерпретатор перед виконанням системного виклику exec відкриває вказані файли, підмінюючи зміст дескрипторів 1, 2, 3.

Файлова система

Файл у Unix – множина символів з довільним доступом. У файлах розміщуються довільні дані. Файл має таку структуру, яку вимагає від нього користувач.

Інформація на дисках розміщується блоками – по 512 байт у кожному блоці. Блок спеціально обрано відповідно до розміру сектора.

Диск поділено на наступні області (рис. 11.1):

- невикористовуваний блок;
- керуючий або суперблок, у якому зберігається розмір диску та розміри інших областей;
- і-список, складений з опису файлів, що називають і-вузлами;
- область для зберігання вмісту файлів.

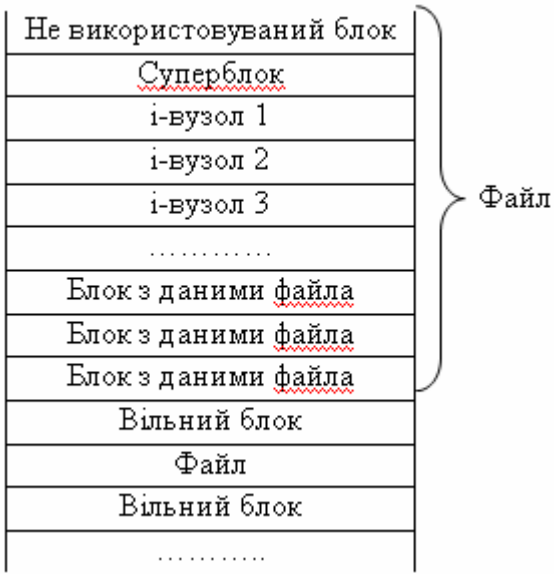


Рис. 11.1. Поділ диску на області

Організація файлів в Unix диску. Кожний вузол містить:

- ідентифікацію користувача;
- ідентифікацію групи власника;
- біти захисту;
- фізичні адреси на диску, де знаходиться вміст файлу;
- розмір файлу;
- час створення файлу;
- час останнього використання файлу;
- час останньої змінення атрибутів;
- кількість зв’язок-вказівників, що вказують на файл;
- індикацію, що являє собою файл-директорію, і є звичайним файлом чи спеціальним файлом.

За і-списком йдуть блоки, що визначені для зберігання вмісту файлів. Простір на диску, що залишився вільним від файлів, створює зв’язаний список вільних блоків.

Файлова система – Unix структура даних, яка має у своєму складі керуючий суперблок, у якому визначена файлова система в цілому; масив і-вузлів, де визначені всі файли, самі файли та сукупність вільних блоків. Виділення простору під дані виконується блоками фіксованого розміру.

Кожен файл однозначно ідентифікується старшим номером пристрою, молодшим номером пристрою та і-номером (індексом і-вузла даного файлу у масиві і-вузлів). Коли викликають драйвер пристрою, за старшим номером індексується масив вхідних точок в драйвери. За молодшим номером, драйвер обирає один пристрій з групи ідентичних фізичних пристроїв. Файл-директорія, у якому перераховані імена файлів, дозволяє встановити відповідність між іменами та самими файлами. Директорії створюють деревоподібну структуру. На кожний звичайний файл або файл-пристрій може бути посилання у різних вузлах цієї структури. У непривілейованих програмах запис в директорії не дозволено, але при наявності відповідних дозволів вони можуть читати їх. Додаткових зв’язків між директивами немає.

Велику кількість системних директорій ОС використовує для своїх власних потреб. Одна з них – коренева директорія є базою для всієї структури директорій, і йдучи від неї, можна знайти розміщення всіх файлів. В інших системних директоріях знаходяться програми та команди, що надаються користувачам, та файли пристроїв. Імена файлів задаються послідовністю імен директорій.

Файл, що не є директорією, може зустрічатися у різних директоріях, можливо під різними іменами. Це має назву **зв'язування.** Елемент в директорії, що відноситься до одного файлу, звуть зв'язком. У ОС Unix всі такі зв'язки мають однаковий статус. Файли же належать до директорій. Вони існують незалежно від елементів директорій, а зв'язки в директоріях вказують на фізичні файли. Файл зникає, коли зникає останній зв'язок, що вказує на нього.

Захист файлів

Захист файлів виконується за допомогою номеру, що ідентифікує користувача, та встановлення десяти бітів захисту – атрибутів доступу. Права доступу поділяють на три типи: читання (read), запис (write) та виконання (execute). Ці типи прав доступу можуть бути надані трьом класам користувачів: власнику файла, групі, у яку входить власник, та всім іншим користувачам. Дев'ять з цих бітів керують захистом щодо читання, запису і виконання для власника файлу, інших членів групи, у яку входить власник, та всіх інших користувачів. Файл завжди зв'язаний з конкретним користувачем – своїм власником – і з визначеною групою, тобто у нього вже є відоме UID (user ID, ідентифікатор користувача) та GID (group ID, ідентифікатор групи).

Змінити права доступу до файлу може тільки його власник, змінити власника файлу може суперкористувач, групу – суперкористувач та власник файлу.

Виконувана в системі програма завжди запускається від імені якогось користувача та деякої групи, але зв'язок процесів із користувачами та групами організований складніше: тут розрізняють ідентифікатор доступу до файлової системи (FSUID – file system access user ID, FSGID – file system access group ID) та ефективний ідентифікатор (EUID – effective user ID, EGID – effective group ID), а при доступі до файлу враховуються ще й повноваження (capabilities), приписані самому процесу.

При створенні файл отримує UID (ідентифікатор користувача), що співпадає з FSUID (ідентифікатор доступу до файлової системи) процесу, який його створює, і як правило GID (ідентифікатор групи), що співпадає з FSGID (ідентифікатор групи доступу до файлової системи).

Атрибути доступу визначають, що дозволено робити з конкретним файлом даної категорії користувачів. Мають місце всього три операції: читання, запис, виконання.

При створенні файлу (або ще одного імені для вже існуючого файлу) модифікується не сам файл, а каталог, у якому з'являються нові посилання на вузли. Знищення файлу – це знищення посилання. Тобто, право на створення чи знищення файлу – це право на запис у каталог.

Право на виконання каталогу інтерпретується, як право на пошук у ньому (власне, проходження через нього). Воно дозволяє звернутись до файлу на шляху, що вміщує даний каталог, навіть, тоді, коли каталог не дозволено читати, а тому список всіх його файлів є недоступним.

Окрім трьох названих основних атрибутів існують додаткові, що використовуються у таких випадках.

Атрибути SUID та SGID є суттєвими при запуску програми на виконання: вони вимагають, щоб програма виконувалась не від імені користувача, що запустив програму, а від імені власника (або групи) того файлу, в якій вона знаходиться. Завдяки цьому, користувачі отримують можливість запускати системну програму, яка створює свої робочі файли у закритих для них каталогах.

Окрім того, якщо процес створює файл у каталозі, що має атрибути SGID, то файл отримує GID не по FSGID, а по GID каталогу. Це зручно для колективної роботи: усі файли і підкаталоги у каталозі автоматично прив'язуються до однієї й тієї ж групи, хоча створювати їх можуть різні користувачі.

І ще один атрибут CVTX, який відноситься до каталогів. Він показує, з каталогу, що має атрибут, посилання на файл може знищити тільки власник файлу.

Існують дві стандартні форми запису до прав доступу: символьна та вісімкова. Символьна – це ланцюг з десяти знаків. Вісімковий запис – це шестизначне число, перші два знаки якого визначають тип файлу, далі атрибути, останні три – права власника, групи та всіх інших.

Міжпроцесорні комунікації

ОС Unix повністю відповідає технології «клієнт-сервер». Ця універсальна модель слугує основою побудови систем довільної складності, в тому числі й мережевих. Для цього в ОС існують наступні механізми:

- сигнали;
- семафори;
- програмні канали;
- черги повідомлень;
- сегменти поділюваної пам'яті;
- виклики віддалених процедур.

Багато з цих засобів вже добре відомі, тому розглянемо їх досить коротко.

Сигнали. Якщо розглядати виконання процесу у віртуальному комп'ютері, який надається кожному користувачеві, то в такій системі повинна існувати система переривань, що відповідає наступним вимогам:

- обробка виняткових ситуацій;
- засоби обробки зовнішніх та внутрішніх переривань;

- засоби управління системою переривань (маскування та демаскування).

ОС здатна не тільки приймати та обробляти сигнали, але й породжувати їх та посилати на інші машини або процеси. Сигнали можуть бути синхронними, коли ініціатором сигналу є сам процес, та асинхронними, коли ініціатором є користувач за терміналом. Джерелом асинхронних сигналів може бути також ядро, коли воно контролює визначені стани апаратури, що розглядаються як помилкові.

Сигнали можна розглядати, як найпростішу форму міжпроцесної взаємодії, що повідомляють процеси або ядро про виникнення конкретних подій.

Семафори. Механізм семафорів – узагальнення класичного механізму семафорів, які розглядались раніше.

Семафор в ОС Unix складається з таких елементів:

- значення семафору;
- ідентифікатор процесу, який хронологічно останнім працював з семафором;
- кількість процесів, що чекають збільшення значення семафору;
- кількість процесів, що чекають нульового значення.

Для роботи з семафорами є наступні системні виклики:

- semget – для створення та отримання доступу;
- semop – для маніпуляції значеннями семафорів (використовується для синхронізації процесів);
- semctl – для виконання різних керуючих операцій над набором семафорів.

Основним системним викликом для маніпуляції семафором є semop:

```
Oldval = semop (id, oplist, count);
```

id – раніше отриманий дескриптор групи семафорів;

oplist – масив описувачів операцій над семафорами групи;

count – розмір цього масиву.

Значення, що повертаються системним викликам, є значення останнього опрацьованого семафора. Кожний з елементів oplist має наступну структуру:

- номер семафору у вказаному наборі семафорів;
- операція;
- ознаки.

Якщо перевірка прав доступу проходить нормально, та вказані в масиві oplist номера семафорів не виходять за межі розміру набору семафорів, то системний виклик виконується наступним чином. Для кожного елементу масиву oplist значення відповідного семафору змінюється у відповідності до значення поля «операція»:

- якщо значення поля позитивне, то значення семафору збільшується на одиницю, а всі процеси, що чекають збільшення значення семафору, активізуються (або «пробуджуються» в термінах Unix);
- якщо значення поля операції дорівнює нулю, то, якщо значення семафору теж дорівнює нулю, обирається наступний елемент масиву oplist. Якщо ж значення семафору відмінне від нуля, то ядро збільшує на одиницю кількість процесів, що очікують нульового значення семафору, а той процес, що звернувся, переходить в стан очікування («усипляється» в термінології Unix).
- Якщо значення поля операції від'ємне, та його абсолютне значення менше чи дорівнює значенню семафора, то ядро додає це від'ємне значення до значення семафора. Якщо в результаті значення семафору стало нульовим, то ядро активізує (пробуджує) всі процеси, що чекали нульового значення цього семафору. Якщо ж значення семафору менше абсолютної величини поля операції, то ядро збільшує на одиницю кількість процесів, що очікують збільшення значення семафору та відкладає («присипляє») поточний процес до настання цієї події.

Цікаво, що приводом для введення масових операцій над семафорами, як пояснюють розробники Unix, було бажання уникати тупикових ситуацій.

Програмні канали. Програмні канали (pipe) в ОС Unix є важливим засобом взаємодії та синхронізації процесів. Теоретично програмний канал дозволяє взаємодіяти довільній кількості процесів, забезпечуючи дисципліну FIFO (стек). Тобто процес, що читає з програмного каналу, прочитає найдавніші дані, що записані в програмний канал. При традиційній реалізації програмних каналів для збереження даних використовувались файли. У сучасних версіях Unix для реалізації програмних каналів застосовуються інші засоби IPC (зокрема черги повідомлень).

Розрізняють два види програмних каналів: іменовані і неіменовані. Іменований може слугувати для сполучення та синхронізації довільних процесів, що знають ім'я програмного каналу та мають відповідні права доступу.

Неіменованим програмним каналом може користуватися лише процес, який створив цей канал, та його нащадки (не обов'язково прямі).

Для створення іменованого програмного каналу (або отримання до нього доступу) використовується звичайний системний файловий виклик open.

Для створення неіменованого програмного каналу існує спеціальний системний виклик pipe. Однак після отримання відповідних дескрипторів обидва види програмних каналів використовуються однаково за допомогою стандартних файлових системних викликів read, write та close.

Системний виклик pipe має наступний синтаксис:

```
pipe (fdptr);
```

де fdptr – вказівник на масив з двох цілих чисел, в які після створення неіменованого програмного каналу будуть записані дескриптори, визначені для читання з програмного каналу (за допомогою системного виклику read) та запису у програмний канал (за допомогою системного виклику write). Дескриптори неіменованого програмного каналу – це звичайні дескриптори

файлів, тобто такому програмному каналу відповідають два елементи таблиці відкритих файлів процесу. Тому при наступному використанні системних викликів read та write процес не відрізняє випадки використання програмних каналів від випадка використання звичайних файлів.

Для створення іменованих програмних каналів (або для отримання доступу до вже існуючих) використовується звичайний системний виклик open. Основною відмінністю від випадку відкриття звичайного файлу є те, що коли іменований програмний канал відкривається для запису та жоден з процесів не відкриває той самий канал для читання, то процес, що звернувся, блокується до тих пір, доки деякий процес не відкриє деякий програмний канал для читання. Аналогічно виконується обробка і читання. Закінчення роботи процесу з програмним каналом виконується за допомогою системного виклику close.

Черги повідомлень. Для забезпечення можливості обміну повідомленнями між процесами цей механізм підтримується наступними системними викликами:

- msgget для створення нової черги повідомлень або отримання дескриптора існуючої черги;
- msgsnd для відправлення повідомлення (точніше, для постановки його у вказану чергу);
- msgrev для отримання повідомлення (точніше, для виборки повідомлення з черги повідомлень);
- msgct для виконання ряду керуючих функцій.

Системний виклик msgget має стандартний для сімейства get синтаксис:

```
msggid = msgget (key, flag)
```

При виконанні цього виклику ОС або створює нову чергу повідомлень, поміщаючи його в заголовок таблиці черг повідомлень та повертаючи користувачу дескриптор заново створеної черги, або знаходить елемент

таблиці черг повідомлень, які мають вказаний ключ та повертають відповідний дескриптор черги.

Не будемо далі розшифровувати синтаксис інших системних викликів, бо це не є поточною метою нашого дослідження.

Поділювана пам'ять. Для роботи з поділюваною пам'яттю використовуються чотири системні виклики:

- shmget – створює новий сегмент поділюваної пам'яті або знаходить існуючий сегмент з тим же ключем;
- shmat – підключає сегмент із вказаним дескриптором до віртуальної пам'яті процесу, що звертається;
- shmdt – відключає від віртуальної пам'яті підключений до неї сегмент за вказаною початковою віртуальною адресою;
- shmct – слугує для управління різними параметрами, пов'язаними з існуючим сегментом.

Після того як сегмент поділюваної пам'яті підключено до віртуальної пам'яті процесу, він може звертатися до відповідних елементів пам'яті за допомогою звичайних машинних команд читання та запису без звернення до додаткових системних викликів.

Виклики віддалених процедур (RPC). У багатьох випадках, взаємодія процесів носить характер «клієнт-сервер». Один з процесів запитує послугу і не продовжує свого виконання до тих пір, поки не отримає цю послугу. Сегментно такий режим дуже схожий до виклику процедури. Звідти й відповідна назва. ОС Unix ідеально підходить для того, щоб бути мережевою операційною системою. Однак залишається проблема різного представлення даних у комп'ютерах з різною архітектурою. Тому основною ідеєю RPC є автоматичне забезпечення перетворення форматів даних при взаємодії процесів, що виконуються на різних комп'ютерах.

Реалізація технології віддалених процедур досить складна, оскільки цей механізм має забезпечити взаємодію процесів, що реалізуються на різних комп'ютерах. Якщо у випадку звернення до процедури у тому ж самому комп'ютері, процес використовує стек або загальні області пам'яті, то при віддаленому виклику передача параметрів процедури перетворюється у передачу запиту мережею, а це вже інша проблема.

11.2 Операційна система LINUX

Це подібна до Unix ОС. Це багатозадачна та багатокористувацька ОС. Вона достатньо добре сумісна з Unix на рівні вхідних текстів. Більшість програм, орієнтованих на Unix, придатна для Linux без змін. Linux підтримує різні типи файлових систем для зберігання даних. Реалізована також система управління файлами на базі FAT, що дозволяє безпосередньо звертатись до файлів, що знаходяться у цій файловій системі. Підтримується файлова система ISO 9660 CD-ROM для роботи з дисками CD-ROM. Система управління файлами FAT32 теж може бути використана.

Linux забезпечує повний набір протоколів TCP/IP для мережевої роботи.

Ядро Linux було створено з урахуванням можливостей захищеного режиму процесорів 80386 та 80486. Linux підтримує завантаження тільки потрібних сторінок. Є можливість використання однієї сторінки, фізично один раз завантаженої в пам'ять одразу декількома процесами.

Програми, що виконуються, використовують динамічно зв'язані бібліотеки, тобто програми, що виконуються, мають можливість сумісно використовувати бібліотечну програму, що представлена одним файлом на диску. Це дозволяє виконуваним файлам займати менше місця на диску, особливо тим файлам, які багаторазово використовують бібліотечні функції.

Є можливість також використовувати статично зв'язані бібліотеки, якщо користувач бажає користуватись відлагодженням на рівні об'єктних кодів або мати «повні» програми, що не потребують бібліотек, що поділяються.

Linux дозволяє бібліотекам, що поділяються, динамічно зв’язуватися під час виконання, що дозволяє програмісту заміняти бібліотечні модулі своїми власними.

11.3 Контрольні запитання до розділу 11

1. Якою мовою програмування розроблена ОС UNIX?
2. Що визначає поняття образ та які складові образу?
3. Які особливості логічної та фізичної побудови файлової системи ОС UNIX?
4. Які засоби міжпроцесної комунікації використовує ОС UNIX?
5. Які особливості застосування семафорів і каналів в ОС UNIX?
6. Що означае термін виклик віддалених процедур і як реалізується така процедура?
7 Які особливості побудови і використання ОС LINUX?
8. Яким чином у ОС UNIX забезпечується робота з поділюваною пам’ятью?

ЛІТЕРАТУРА

Основна література

1. Шеховцов В. А. Операційні системи / В. А. Шеховцов – К.: Вид. гр. BHV, 2005. – 576 с.

Допоміжна література

2. Олифер В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер – СПб: Питер, 2001. – 544 с.

3. Цикридис Д. Операционные системы / Д.Цикридис, Ф. Бернстайн – М: – Мир, 1977.

4. Иртегов Д. Введение в операционные системы / Д. Иртегов – СПб.: БХВ – Петербург, 2002.

5. Абрамович С.М. Технология программирования: методы и средства / С.М. Абрамович – Изд. Ростовского университета, 1992.

6. Лингер Р. Теория и практика структурного программирования / Р. Лингер, Х. Миллс, Б. Уитт – М.: Мир, 1982.