Лекція 1

Вступ.

Що таке операційна система

Структура обчислювальної системи

З чого складається будь-яка обчислювальна система? По-перше, з того, що в англомовних країнах прийнято називати словом hardware, або апаратне забезпечення: процесор, пам'ять, монітор, дискові пристрої і так далі, об'єднані магістральним з'єднанням, яке називається шиною.

По-друге, обчислювальна система складається з програмного забезпечення. Все програмне забезпечення прийнято ділити на дві частини: прикладне і системне. До прикладного програмного забезпечення, як правило, відносяться різноманітні банківські та інші бізнес-програми, ігри, текстові процесори і таке інше. Під системним програмним забезпеченням зазвичай розуміють програми, що сприяють функціонуванню і розробці прикладних програм. Треба сказати, що поділ на прикладне і системне програмне забезпечення є доволі умовним і залежить від того, хто здійснює такий поділ. Так, звичайний користувач, недосвідчений в програмуванні, може вважати Microsoft Word за системну програму, а, з погляду програміста, це – застосування. Компілятор мови С для звичайного програміста – системна програма, а для системного – прикладна. Не зважаючи на таку нечітку грань, цю ситуацію можна відобразити у вигляді послідовності шарів (див. рис. 1.1), виділивши окремо найбільш загальну частину системного програмного забезпечення – операційну систему:

Рис. 1.1. Шари програмного забезпечення комп'ютерної системи

Що таке ОС

Більшість користувачів мають досвід експлуатації операційних систем, проте їм важко дати цьому поняттю точне визначення. Давайте коротко розглянемо основні точки зору.

Операційна система як віртуальна машина

При розробці ОС широко застосовується абстрагування, яке є важливим методом спрощення і дозволяє концентруватися на взаємодії високорівневих компонентів системи, ігноруючи деталі їх реалізації. У цьому сенсі ОС є інтерфейсом між користувачем і комп'ютером.

Архітектура більшості комп'ютерів на рівні машинних команд дуже незручна для використання прикладними програмами. Наприклад, робота з диском припускає знання внутрішньої будови його електронного компоненту – контролера, для введення команд обертання диска, пошуку і форматування доріжок, читання і запису секторів і т.д. Зрозуміло, що середній програміст не в змозі враховувати всі особливості роботи обладнання (в сучасній термінології – займатися розробкою драйверів пристроїв), а повинен мати просту високорівневу абстракцію, яка, наприклад, представляє інформаційний простір диска як набір файлів. Файл можна відкривати для читання або запису, використовувати для отримання або збереження інформації, а потім закривати. Це концептуально простіше, ніж дбати про деталі переміщення головок дисків або організації роботи мотора. Аналогічним чином, за допомогою простих і ясних абстракцій, ховаються від програміста всі непотрібні подробиці організації переривань, роботи таймера, управління пам'яттю і т.д. Більш того, на сучасних обчислювальних комплексах можна створити ілюзію необмеженого розміру оперативної пам'яті і кількості процесорів. Всім цим займається операційна система. Таким чином, операційна система представляється користувачеві віртуальною машиною, з якою простіше мати справу, ніж безпосередньо з обладнанням комп'ютера.

Операційна система як менеджер ресурсів

Операційна система призначена для управління всіма частинами вельми складної архітектури комп'ютера. Представимо, наприклад, що відбудеться, якщо декілька програм, що працюють на одному комп'ютері, намагатимуться одночасно здійснювати вивід на принтер. Ми отримали б мішанину строчок і сторінок, виведених різними програмами. Операційна система запобігає такого роду хаосу за рахунок буферизації інформації, призначеної для друку, на диску і організації черги на друк. Для багатокористувацьких комп'ютерів необхідність управління ресурсами і їх захисту ще очевидніша. Отже, операційна система, як менеджер ресурсів, здійснює впорядкований і контрольований розподіл процесорів, пам'яті і інших ресурсів між різними програмами.

Операційна система як захисник користувачів і програм

Якщо обчислювальна система допускає спільну роботу декількох користувачів, то виникає проблема організації їх безпечної діяльності. Необхідно забезпечити збереження інформації на диску, щоб ніхто не міг видалити або пошкодити чужі файли. Не можна дозволити програмам одних користувачів довільно втручатися в роботу програм інших користувачів. Потрібно присікати спроби несанкціонованого використання обчислювальної системи. Всю цю діяльність здійснює операційна система як організатор безпечної роботи користувачів і їх програм. З такої точки зору операційна система представляється системою безпеки держави, на яку покладені поліцейські і контр розвідувальні функції.

Операційна система як постійно функціонуюче ядро

Нарешті, можна дати і таке визначення: операційна система – це програма, що постійно працює на комп'ютері і взаємодіє зі всіма прикладними програмами. Здавалося б, це абсолютно правильне визначення, але, як ми побачимо далі, в багатьох сучасних операційних системах постійно працює на комп'ютері лише частина операційної системи, яку прийнято називати її ядром.

Як ми бачимо, існує багато точок зору на те, що таке операційна система. Неможливо дати їй адекватне строге визначення. Нам простіше сказати не що є операційна система, а для чого вона потрібна і що вона робить. Для з'ясування цього питання розглянемо історію розвитку обчислювальних систем.

Коротка історія еволюції обчислювальних систем

Ми розглядатимемо історію розвитку саме обчислювальних, а не операційних систем, тому що hardware і програмне забезпечення еволюціонували спільно, роблячи взаємний вплив один на одного. Поява нових технічних можливостей приводила до прориву в області створення зручних, ефективних і безпечних програм, а свіжі ідеї в програмній області стимулювали пошуки нових технічних рішень. Саме ці критерії – зручність, ефективність і безпека – грали роль чинників природного відбору при еволюції обчислювальних систем.

Перший період (1945–1955 рр.). Лампові машини. Операційних систем немає

Ми почнемо дослідження розвитку комп'ютерних комплексів з появи електронних обчислювальних систем (опускаючи історію механічних і електромеханічних пристроїв).

Перші кроки в області розробки електронних обчислювальних машин були зроблені в кінці Другої світової війни. В середині 40-х були створені перші лампові обчислювальні пристрої і з'явився принцип програми, що зберігається в пам'яті машини (John Von Neumann, червень 1945 р.). У той час одна і та ж група людей брала участь і в проектуванні, і в експлуатації, і в програмуванні обчислювальної машини. Це була швидшою науково-дослідна робота в області обчислювальної техніки, а не регулярне використання комп'ютерів як інструмент вирішення яких-небудь практичних завдань з інших прикладних областей. Програмування здійснювалося виключно на машинній мові. Про операційні системи не було і мови, всі завдання організації обчислювального процесу вирішувалися уручну кожним програмістом з пульта управління. За пультом міг знаходитися тільки один користувач. Програма завантажувалася в пам'ять машини в кращому разі з колоди перфокарт, а зазвичай за допомогою панелі перемикачів.

Обчислювальна система виконувала одночасно тільки одну операцію (уведення-виведення або власне обчислення). Відладка програм велася з пульта управління за допомогою вивчення стану пам'яті і регістрів машини. В кінці цього періоду з'являється перше системне програмне забезпечення: у 1951–1952 рр. виникають прообрази перших компіляторів з символічних мов (Fortran і ін.), а в 1954 р. Nat Rochester розробляє Асемблер для IBM-701.

Істотна частина часу йшла на підготовку запуску програми, а самі програми виконувалися строго послідовно. Такий режим роботи називається послідовною обробкою даних. В цілому перший період характеризується украй високою вартістю обчислювальних систем, їх малою кількістю і низькою ефективністю використання.

Другий період (1955 грама.–начало 60-х). Комп'ютери на основі транзисторів. Пакетні операційні системи

З середини 50-х років почався наступний період в еволюції обчислювальної техніки, пов'язаний з появою нової технічної бази – напівпровідникових елементів. Застосування транзисторів замість часто перегораючих електронних ламп привело до підвищення надійності комп'ютерів. Тепер машини можуть безперервно працювати достатньо довго, щоб на них можна було покласти виконання практично важливих завдань. Знижується споживання обчислювальними машинами електроенергії, удосконалюються системи охолоджування. Розміри комп'ютерів зменшилися. Знизилася вартість експлуатації і обслуговування обчислювальної техніки. Почалося використання ЕОМ комерційними фірмами. Одночасно спостерігається бурхливий розвиток алгоритмічних мов (LISP, COBOL, ALGOL-60, PL-1 і так далі). З'являються перші справжні компілятори, редактори зв'язків, бібліотеки математичних і

службових підпрограм. Спрощується процес програмування. Пропадає необхідність звалювати на одних і тих же людей весь процес розробки і використання комп'ютерів. Саме у цей період відбувається розділення персоналу на програмістів і операторів, фахівців по експлуатації і розробників обчислювальних машин.

Змінюється сам процес прогону програм. Тепер користувач приносить програму з вхідними даними у вигляді колоди перфокарт і указує необхідні ресурси. Така колода отримує назву завдання. Оператора завантажує завдання в пам'ять машини і запускає його на виконання. Отримані вихідні дані друкуються на принтері, і користувач отримує їх назад через деякий час.

Зміна запитаних ресурсів викликає припинення виконання програм, в результаті процесор часто простоює. Для підвищення ефективності використання комп'ютера завдання з схожими ресурсами починають збирати разом, створюючи пакет завдань.

З'являються перші системи пакетної обробки, які просто автоматизують запуск однієї програми з пакету за іншою і тим самим збільшують коефіцієнт завантаження процесора. При реалізації систем пакетної обробки була розроблена формалізована мова управління завданнями, за допомогою якого програміст повідомляв систему і оператора, яку роботу він хоче виконати на обчислювальній машині. Системи пакетної обробки стали прообразом сучасних операційних систем, вони були першими системними програмами, призначеними для управління обчислювальним процесом.

Третій період (почало 60-х – 1980 р.). Комп'ютери на основі інтегральних мікросхем. Перші багатозадачні ОС

Наступний важливий період розвитку обчислювальних машин відноситься на початок 60-х – 1980 р. В цей час в технічній базі відбувся перехід від окремих напівпровідникових елементів типу транзисторів до інтегральних мікросхем. Обчислювальна техніка стає надійнішою і дешевшою. Росте складність і кількість завдань, що вирішуються комп'ютерами. Підвищується продуктивність процесорів.

Підвищенню ефективності використання процесорного часу заважає низька швидкість роботи механічних пристроїв введення-виводу (швидкий считыватель перфокарт міг обробити 1200 перфокарт в хвилину, принтери друкували до 600 рядків в хвилину). Замість безпосереднього читання пакету завдань з перфокарт в пам'ять починають використовувати його попередній запис, спочатку на магнітну стрічку, а потім і на диск. Коли в процесі виконання завдання потрібне введення даних, вони читаються з диска. Так само вихідна інформація спочатку копіюється в системний буфер і записується на стрічку або диск, а друкується тільки після завершення завдання. Спочатку дійсні операції введення-виводу здійснювалися в режимі off-line, тобто з використанням інших, простіших, окремо вартих комп'ютерів. Надалі вони починають виконуватися на тому ж комп'ютері, який проводить обчислення, тобто в режимі on-line. Такий прийом отримує назва spooling (скорочення від Simultaneous Peripheral Operation On Line) або підкачки-відкачування даних. Введення техніки підкачки-відкачування в пакетні системи дозволило сумістити реальні операції введення-виводу одного завдання з виконанням іншого завдання, але зажадало розробки апарату переривань для сповіщення процесора про закінчення цих операцій.

Магнітні стрічки були пристроями послідовного доступу, тобто інформація прочитувалася з них в тому порядку, в якому була записана. Поява магнітного диска, для якого не важливий порядок читання інформації, тобто пристрої прямого доступу, привела до подальшого розвитку обчислювальних систем. При обробці пакету завдань на магнітній стрічці черговість запуску завдань визначалася порядком їх введення. При обробці пакету завдань на магнітному диску з'явилася можливість вибору чергового виконуваного завдання. Пакетні системи починають

займатися плануванням завдань: залежно від наявності запитаних ресурсів, терміновості обчислень і так далі на рахунок вибирається те або інше завдання.

Подальше підвищення ефективності використання процесора було досягнуте за допомогою мультипрограмування. Ідея мультипрограмування полягає в наступному: поки одна програма виконує операцію введення-виводу, процесор не простоює, як це відбувалося при однопрограммному режимі, а виконує іншу програму. Коли операція введення-виводу закінчується, процесор повертається до виконання першої програми. Ця ідея нагадує поведінку викладача і студентів на іспиті. Поки один студент (програма) обдумує відповідь на питання (операція введення-виводу), викладач (процесор) вислуховує відповідь іншого студента (обчислення). Природно, така ситуація вимагає наявність в кімнаті декількох студентів. Так само мультипрограмування вимагає наявність в пам'яті декількох програм одночасно. При цьому кожна програма завантажується в свою ділянку оперативної пам'яті, звану розділом, і не повинна впливати на виконання іншої програми. (Студенти сидять за окремими столами і не підказують один одному.)

Поява мультипрограмування вимагає справжньої революції в будові обчислювальної системи. Особливу роль тут грає апаратна підтримка (багато апаратних нововведень з'явилися ще на попередньому етапі еволюції), найбільш істотні особливості якої перераховані нижче.

- Реалізація захисних механізмів. Програми не повинні мати самостійного доступу до розподілу ресурсів, що приводить до появи привілейованих і непривілейованих команд. Привілейовані команди, наприклад команди введення-виводу, можуть виконуватися тільки операційною системою. Говорять, що вона працює в привілейованому режимі. Перехід управління від прикладної програми до ОС супроводиться контрольованою зміною режиму. Крім того, це захист пам'яті, що дозволяє ізолювати конкуруючі призначені для користувача програми один від одного, а ОС – від програм користувачів.
- Наявність переривань. Зовнішні переривання оповіщають ОС про те, що відбулося асинхронна подія, наприклад завершилася операція введення-виводу. Внутрішні переривання (зараз їх прийнято називати винятковими ситуаціями) виникають, коли виконання програми привело до ситуації, що вимагає втручання ОС, наприклад ділення на нуль або спроба порушення захисту.
- Розвиток паралелізму в архітектурі. Прямий доступ до пам'яті і організація каналів введення-виводу дозволили звільнити центральний процесор від рутинних операцій.

Не менш важлива в організації мультипрограмування роль операційної системи. Вона відповідає за наступні операції.

- Організація інтерфейсу між прикладною програмою і ОС за допомогою системних викликів.
- Організація черги із завдань в пам'яті і виділення процесора одному із завдань зажадало планування використання процесора.
- Перемикання з одного завдання на інше вимагає збереження вмісту регістрів і структур даних, необхідних для виконання завдання, інакше кажучи, контексту для забезпечення правильного продовження обчислень.
- Оскільки пам'ять є обмеженим ресурсом, потрібні стратегії управління пам'яттю, тобто потрібно упорядкувати процеси розміщення, заміщення і вибірки інформації з пам'яті.

- Організація зберігання інформації на зовнішніх носіях у вигляді файлів і забезпечення доступу до конкретного файлу тільки певним категоріям користувачів.
- Оскільки програмам може потрібно провести санкціонований обмін даними, необхідно їх забезпечити засобами комунікації.
- Для коректного обміну даними необхідно вирішувати конфліктні ситуації, що виникають при роботі з різними ресурсами і передбачити координацію програмами своїх дій, тобто забезпечити систему засобами синхронізації.

Мультипрограмні системи забезпечили можливість ефективнішого використання системних ресурсів (наприклад, процесора, пам'яті, периферійних пристроїв), але вони ще довго залишалися пакетними. Користувач не міг безпосередньо взаємодіяти із завданням і повинен був передбачити за допомогою карт, що управляють, всі можливі ситуації. Відладка програм як і раніше займала багато часу і вимагала вивчення багатосторінкових роздруків вмісту пам'яті і регістрів або використання налагоджувального друку.

Поява електронно-променевих дисплеїв і переосмислення можливостей застосування клавіатур поставили на чергу вирішення цієї проблеми. Логічним розширенням систем мультипрограмування сталі time-sharing системи, або системи розділення времени1). У них процесор перемикається між завданнями не тільки на час операцій введення-виводу, але і просто після певного часу. Ці перемикання відбуваються так часто, що користувачі можуть взаємодіяти зі своїми програмами під час їх виконання, тобто інтерактивно. В результаті з'являється можливість одночасної роботи декількох користувачів на одній комп'ютерній системі. У кожного користувача для цього має бути хоч би одна програма в пам'яті. Щоб зменшити обмеження на кількість працюючих користувачів, була упроваджена ідея неповного знаходження виконуваної програми в оперативній пам'яті. Основна частина програми знаходиться на диску, і фрагмент, який необхідно в даний момент виконувати, може бути завантажений в оперативну пам'ять, а непотрібний – викачаний назад на диск. Це реалізується за допомогою механізму віртуальної пам'яті. Основною гідністю такого механізму є створення ілюзії необмеженої оперативної пам'яті ЕОМ.

У системах розділення часу користувач дістав можливість ефективно проводити відладку програми в інтерактивному режимі і записувати інформацію на диск, не використовуючи перфокарти, а безпосередньо з клавіатури. Поява on-line-файлов привела до необхідності розробки розвинених файлових систем.

Паралельно внутрішній еволюції обчислювальних систем відбувалася і зовнішня їх еволюція. До початку цього періоду обчислювальні комплекси були, як правило, несумісні. Кожен мав власну операційну систему, свою систему команд і так далі В результаті програму, що успішно працює на одному типі машин, необхідно було повністю переписувати і наново відладжувати для виконання на комп'ютерах іншого типу. На початку третього періоду з'явилася ідея створення сімейств програмно сумісних машин, що працюють під управлінням однієї і тієї ж операційної системи. Першим сімейством програмно сумісних комп'ютерів, побудованих на інтегральних мікросхемах, стала серія машин IBM/360. Розроблене на початку 60-х років, це сімейство значно перевершувало машини другого покоління по критерію ціна/продуктивність. За ним послідувала лінія комп'ютерів PDP, несумісних з лінією IBM, і кращою моделлю в ній стала PDP-11.

Сила "однієї сім'ї" була одночасно і її слабкістю. Широкі можливості цієї концепції (наявність всіх моделей: від міні-комп'ютерів до гігантських машин; велика кількість різноманітної периферії; різне оточення; різні користувачі) породжували складну і громіздку операційну систему. Мільйони строчок Асемблера, написані тисячами програмістів, містили безліч помилок, що викликало безперервний потік публікацій про них і спроб виправлення.

Тільки у операційній системі OS/360 містилося більше 1000 відомих помилок. Проте ідея стандартизації операційних систем була широко упроваджена в свідомість користувачів і надалі отримала активний розвиток.

Четвертий період (з 1980 р. по теперішній час). Персональні комп'ютери. Класичні, мережеві і розподілені системи

Наступний період в еволюції обчислювальних систем пов'язаний з появою великих інтегральних схем (БІС). У ці роки відбулося різке зростання ступеня інтеграції і зниження вартості мікросхем. Комп'ютер, що не відрізняється по архітектурі від PDP-11, за ціною і простотою експлуатації став доступний окремій людині, а не відділу підприємства або університету. Наступила ера персональних комп'ютерів. Спочатку персональні комп'ютери призначалися для використання одним користувачем в однопрограммном режимі, що спричинило деградацію архітектури цих ЕОМ і їх операційних систем (зокрема, пропала необхідність захисту файлів і пам'яті, планування завдань і т. п.).

Комп'ютери почали використовуватися не тільки фахівцями, що зажадало розробки "дружнього" програмного забезпечення.

Проте зростання складності і різноманітності завдань, що вирішуються на персональних комп'ютерах, необхідність підвищення надійності їх роботи привели до відродження практично всіх рис, характерних для архітектури великих обчислювальних систем.

В середині 80-х стали бурхливо розвиватися мережі комп'ютерів, зокрема персональних, працюючих під управлінням мережевих або розподілених операційних систем.

У мережевих операційних системах користувачі можуть дістати доступ до ресурсів іншого мережевого комп'ютера, тільки вони повинні знати про їх наявність і уміти це зробити. Кожна машина в мережі працює під управлінням своєї локальної операційної системи, що відрізняється від операційної системи автономного комп'ютера наявністю додаткових засобів (програмною підтримкою для мережевих інтерфейсних пристроїв і доступу до видалених ресурсів), але ці доповнення не міняють структуру операційної системи.

Розподілена система, навпаки, зовні виглядає як звичайна автономна система. Користувач не знає і не повинен знати, де його файли зберігаються – на локальній або видаленій машині – і де його програми виконуються. Він може взагалі не знати, чи підключений його комп'ютер до мережі. Внутрішня будова розподіленої операційної системи має істотні відмінності від автономних систем.

Надалі автономні операційні системи ми називатимемо класичними операційними системами.

Проглянувши етапи розвитку обчислювальних систем, ми можемо виділити шість основних функцій, які виконували класичні операційні системи в процесі еволюції:

- Планування завдань і використання процесора.
- Забезпечення програм засобами комунікації і синхронізації.
- Управління пам'яттю.
- Управління файловою системою.
- Управління введенням-виводом.
- Забезпечення безпеки

Кожна з приведених функцій зазвичай реалізована у вигляді підсистеми, що є структурним компонентом ОС. У кожній операційній системі ці функції, звичайно, реалізовувалися по-своєму,

в різному об'ємі. Вони не були спочатку придумані як складові частини операційних систем, а з'явилися в процесі розвитку, у міру того як обчислювальні системи ставали все більш зручними, ефективними і безпечними. Еволюція обчислювальних систем, створених людиною, пішла по такому шляху, але ніхто ще не довів, що це єдино можливий шлях їх розвитку. Операційні системи існують тому, що на даний момент їх існування – це розумний спосіб використання обчислювальних систем. Розгляд загальних принципів і алгоритмів реалізації їх функцій і складає зміст більшої частини нашого курсу, в якому будуть послідовно описані перераховані підсистеми.

Основні поняття, концепції ОС

В процесі еволюції виникло декілька важливих концепцій, які стали невід'ємною частиною теорії і практики ОС. Поняття, що розглядаються в даному розділі, зустрічатимуться і роз'яснюватимуться впродовж всього курсу. Тут дається їх короткий опис.

Системні виклики

У будь-якій операційній системі підтримується механізм, який дозволяє призначеним для користувача програмам звертатися до послуг ядра ОС. У операційних системах найбільш відомої радянської обчислювальної машини БЕСМ-6 відповідні засоби "спілкування" з ядром називалися екстракодами, в операційних системах IBM вони називалися системними макрокомандами і так далі У ОС Unix такі засоби називають системними викликами.

Системні виклики (system calls) – це інтерфейс між операційною системою і призначеною для користувача програмою. Вони створюють, видаляють і використовують різні об'єкти, головні з яких, – процеси і файли. Призначена для користувача програма запрошує сервіс у операційної системи, здійснюючи системний виклик. Є бібліотеки процедур, які завантажують машинні регістри певними параметрами і здійснюють переривання процесора, після чого управління передається обробникові даного виклику, що входить в ядро операційної системи. Мета таких бібліотек – зробити системний виклик схожим на звичайний виклик підпрограми.

Основна відмінність полягає в тому, що при системному виклику завдання переходить в привілейований режим або режим ядра (kernel mode). Тому системні виклики іноді ще називають програмними перериваннями, на відміну від апаратних переривань, які частіше називають просто перериваннями.

У цьому режимі працює код ядра операційної системи, причому виконується він в адресному просторі і в контексті завдання, що викликало його. Таким чином, ядро операційної системи має повний доступ до пам'яті призначеної для користувача програми, і при системному виклику досить передати адреси однієї або декількох областей пам'яті з параметрами виклику і адреси однієї або декількох областей пам'яті для результатів виклику.

У більшості операційних систем системний виклик здійснюється командою програмного переривання (INT). Програмне переривання – це синхронна подія, яка може бути повторене при виконанні однієї і тієї ж програмної коди.

Переривання

Переривання (hardware interrupt) – це подія, що генерується зовнішнім (по відношенню до процесора) пристроєм. За допомогою апаратних переривань апаратура або інформує центральний процесор про те, що відбулося яка-небудь подія, що вимагає негайної реакції (наприклад, користувач натиснув клавішу), або повідомляє про завершення асинхронної операції введення-виводу (наприклад, закінчено читання даних з диска в основну пам'ять). Важливий тип апаратних переривань – переривання таймера, які генеруються періодично через фіксований проміжок часу. Переривання таймера використовуються операційною системою при

плануванні процесів. Кожен тип апаратних переривань має власний номер, однозначно визначальне джерело переривання. Апаратне переривання – це асинхронна подія, тобто воно виникає незалежно від того, який код виконується процесором в даний момент. Обробка апаратного переривання не повинна враховувати, який процес є поточним.

Виняткові ситуації

Виняткова ситуація (exception) – подія, що виникає в результаті спроби виконання програмою команди, яка по якихось причинах не може бути виконана до кінця. Прикладами таких команд можуть бути спроби доступу до ресурсу за відсутності достатніх привілеїв або звернення до відсутньої сторінки пам'яті. Виняткові ситуації, як і системні виклики, є синхронними подіями, що виникають в контексті поточного завдання. Виняткові ситуації можна розділити на поправних і непоправних. До поправних відносяться такі виняткові ситуації, як відсутність потрібної інформації в оперативній пам'яті. Після усунення причини поправної виняткової ситуації програма може виконуватися далі. Виникнення в процесі роботи операційної системи поправних виняткових ситуацій вважається за нормальне явище. Непоправні виняткові ситуації найчастіше виникають в результаті помилок в програмах (наприклад, ділення на нуль). Зазвичай в таких випадках операційна система реагує завершенням програми, що викликала виняткову ситуацію.

Файли

Файли призначені для зберігання інформації на зовнішніх носіях, тобто прийнято, що інформація, записана, наприклад, на диску, повинна знаходитися усередині файлу. Зазвичай під файлом розуміють іменовану частину простору на носієві інформації.

Головне завдання файлової системи (file system) – приховати особливості введення-виводу і дати програмістові просту абстрактну модель файлів, незалежних від пристроїв. Для читання, створення, видалення, запису, відкриття і закриття файлів також є обширна категорія системних викликів (створення, видалення, відкриття, закриття, читання і так далі). Користувачам добре знайомі такі пов'язані з організацією файлової системи поняття, як каталог, поточний каталог, кореневий каталог, шлях. Для маніпулювання цими об'єктами в операційній системі є системні виклики. Файлова система ОС описана в лекціях 11–12.

Процеси, нитки

Концепція процесу в ОС одна з найбільш фундаментальних. Процеси детально розглянуті в лекціях 2–7. Там же описані нитки, або легковагі процеси.

Архітектурні особливості ОС

До цих пір ми говорили про погляд на операційні системи ззовні, про те, що роблять операційні системи. Подальший наш курс буде присвячений тому, як вони це роблять. Але ми поки нічого не сказали про те, що вони є зсередини, які підходи існують до їх побудови.

Монолітне ядро

По суті справи, операційна система – це звичайна програма, тому було б логічне і організувати її так само, як влаштована більшість програм, тобто скласти з процедур і функцій. В цьому випадку компоненти операційної системи є не самостійними модулями, а складовими частинами однієї великої програми. Така структура операційної системи називається монолітним ядром (monolithic kernel). Монолітним ядром є набір процедур, кожна з яких може викликати кожну. Всі процедури працюють в привілейованому режимі. Таким чином, монолітне ядро – це така схема операційної системи, при якій всі її компоненти є складовими частинами однієї програми, використовують загальні структури даних і взаємодіють один з одним шляхом

безпосереднього виклику процедур. Для монолітної операційної системи ядро збігається зі всією системою.

У багатьох операційних системах з монолітним ядром збірка ядра, тобто його компіляція, здійснюється окремо для кожного комп'ютера, на який встановлюється операційна система. При цьому можна вибрати список устаткування і програмних протоколів, підтримка яких буде включена в ядро. Оскільки ядро є єдиною програмою, перекомпіляція – це єдиний спосіб додати в нього нові компоненти або виключити невживані. Слід зазначити, що присутність в ядрі зайвих компонентів украй небажано, оскільки ядро завжди повністю розташовується в оперативній пам'яті. Крім того, виключення непотрібних компонентів підвищує надійність операційної системи в цілому.

Монолітне ядро – старий спосіб організації операційних систем. Прикладом систем з монолітним ядром є більшість Unix-систем.

Навіть у монолітних системах можна виділити деяку структуру. Як у бетонній глибі можна розрізнити вкраплення щебінки, так і в монолітному ядрі виділяються вкраплення сервісних процедур, відповідних системним викликам. Сервісні процедури виконуються в привілейованому режимі, тоді як призначені для користувача програми – в непривілейованому. Для переходу з одного рівня привілеїв на інший іноді може використовуватися головна сервісна програма, що визначає, який саме системний виклик був зроблений, коректність вхідних даних для цього виклику і передавальна управління відповідній сервісній процедурі з переходом в привілейований режим роботи. Іноді виділяють також набір програмних утиліт, які допомагають виконувати сервісні процедури.

Багаторівневі системи (Layered systems)

Продовжуючи структуризацію, можна розбити всю обчислювальну систему на ряд дрібніших рівнів з добре певними зв'язками між ними, так щоб об'єкти рівня N могли викликати тільки об'єкти рівня N-1. Нижнім рівнем в таких системах зазвичай є hardware, верхнім рівнем – інтерфейс користувача. Чим нижче рівень, тим більше привілейовані команди і дії може виконувати модуль, що знаходиться на цьому рівні. Вперше такий підхід був застосований при створенні системи THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) і його студентами в 1968 р. Ця система мала наступні рівні:

Листкова система THE
5. Інтерфейс користувача
4. Управління введенням-виведененям
3. Драйвер пристрою зв'язку оператора і консолі
2. Управління пам'яттю
1. Планування задач та процесів
0. Hardware

Мал. 1.2. Листкова система THE

Листкові системи добре реалізуються. При використанні операцій нижнього шару не потрібно знати, як вони реалізовані, потрібно лише розуміти, що вони роблять. Листкові системи добре тестуються. Відладка починається з нижнього шару і проводиться пошарово. При виникненні помилки ми можемо бути упевнені, що вона знаходиться в тестованому шарі. Листкові системи добре модифікуються. При необхідності можна замінити лише один шар, не

чіпаючи останні. Але листкові системи складні для розробки: важко правильно визначити порядок шарів і що до якого шару відноситься. Листкові системи менш ефективні, чим монолітні. Так, наприклад, для виконання операцій введення-виводу програмі користувача доведеться послідовно проходити всі шари від верхнього до нижнего.

Віртуальні машини

На початку лекції ми говорили про погляд на операційну систему як на віртуальну машину, коли користувачеві немає необхідності знати деталі внутрішнього устрою комп'ютера. Він працює з файлами, а не з магнітними головками і двигуном; він працює з величезною віртуальною, а не обмеженою реальною оперативною пам'яттю; його мало хвилює, єдиний він на машині користувач чи ні. Розглянемо декілька інший підхід. Хай операційна система реалізує віртуальну машину для кожного користувача, але не спрощуючи йому життя, а, навпаки, ускладнюючи. Кожна така віртуальна машина предстає перед користувачем як голе залізо – копія всього hardware в обчислювальній системі, включаючи процесор, привілейовані і непривілейовані команди, пристрої введення-виводу, переривання і так далі І він залишається з цим залізом один на один. При спробі звернутися до такого віртуального заліза на рівні привілейованих команд насправді відбувається системний виклик реальної операційної системи, яка і проводить всі необхідні дії. Такий підхід дозволяє кожному користувачеві завантажити свою операційну систему на віртуальну машину і робити з нею все, що душа побажає.

Програми користувача	Програми користувача	Програми користувача
MS-DOS	Linux	Windows-NT
Віртуальне Hardware	Віртуальне Hardware	Віртуальне Hardware
Реальна операційна система		
Реальне Hardware		

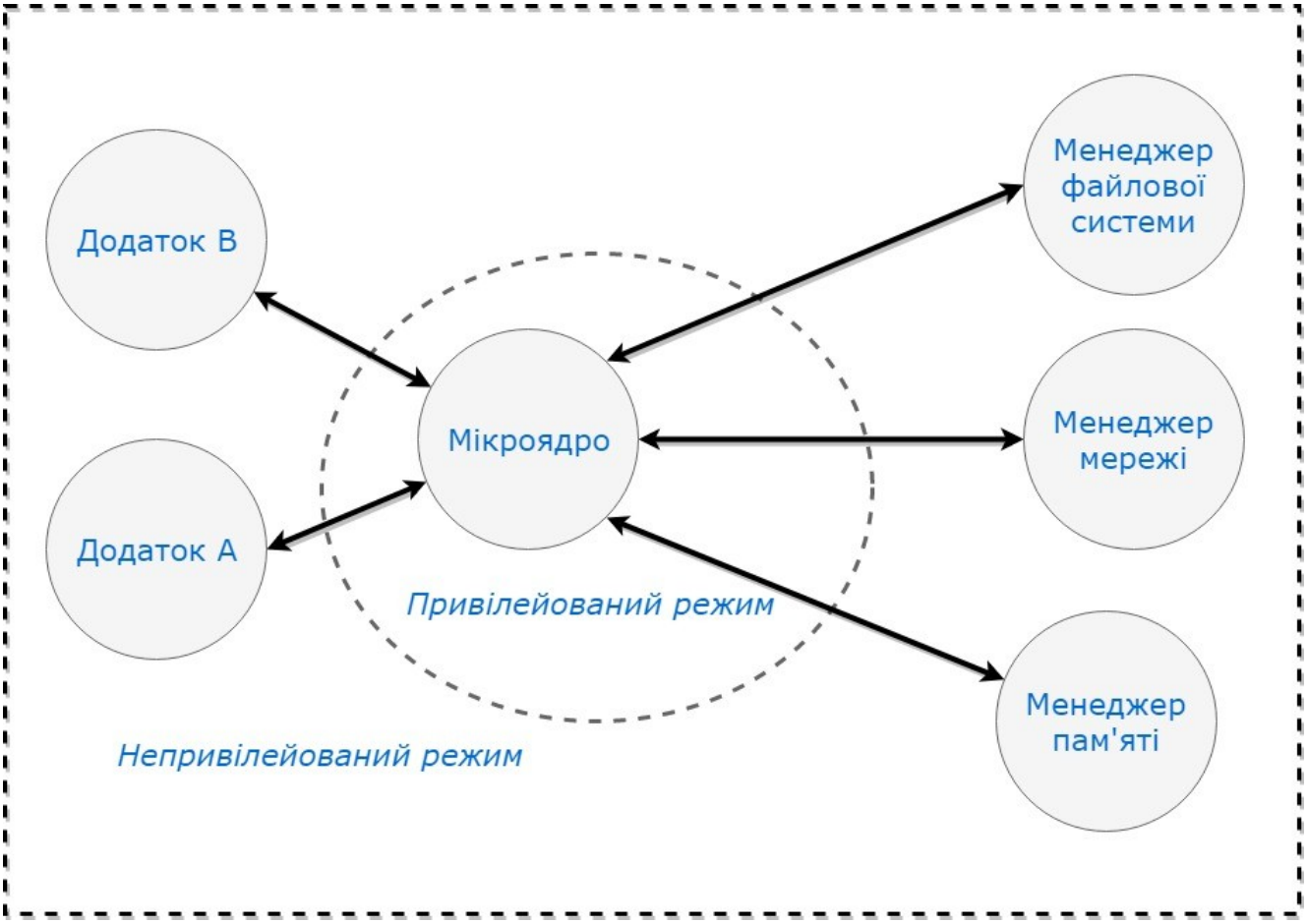
Мал. 1.3. Варіант віртуальної машини

Першою реальною системою такого роду була система CP/CMS, або VM/370, як її називають зараз, для сімейства машин IBM/370.

Недоліком таких операційних систем є зниження ефективності віртуальних машин в порівнянні з реальним комп'ютером, і, як правило, вони дуже громіздкі. Перевага ж полягає у використанні на одній обчислювальній системі програм, написаних для разных операційних систем.

Мікроядерна архітектура

Сучасна тенденція в розробці операційних систем полягає в перенесенні значної частини системної коди на рівень користувача і одночасної мінімізації ядра. Мова йде про підході до побудови ядра, званому мікроядерною архітектурою (microkernel architecture) операційної системи, коли більшість її складових є самостійними програмами. В цьому випадку взаємодія між ними забезпечує спеціальний модуль ядра, званий мікроядром. Мікроядро працює в привілейованому режимі і забезпечує взаємодію між програмами, планування використання процесора, первинну обробку переривань, операції введення-виводу і базове управління пам'яттю.



Мал. 1.4. Мікроядерна архітектура операційної системи

Решта компонентів системи взаємодіє один з одним шляхом передачі повідомлень через мікроядро.

Основна гідність мікроядерної архітектури – високий ступінь модульності ядра операційної системи. Це істотно спрощує додавання в нього нових компонентів. У мікроядерній операційній системі можна, не перериваючи її роботи, завантажувати і вивантажувати нові драйвери, файлові системи і так далі Істотно спрощується процес відладки компонентів ядра, оскільки нова версія драйвера може завантажуватися без перезапуску всієї операційної системи. Компоненти ядра операційної системи нічим принципово не відрізняються від призначених для користувача програм, тому для їх відладки можна застосовувати звичайні засоби. Мікроядерна архітектура підвищує надійність системи, оскільки помилка на рівні непривілейованої програми менш небезпечна, чим відмова на рівні режиму ядра.

В той же час мікроядерна архітектура операційної системи вносить додаткові накладні витрати, пов'язані з передачею повідомлень, що істотно впливає на продуктивність. Для того, щоб мікроядерна операційна система за швидкістю не поступалася операційним системам на базі монолітного ядра, потрібний дуже акуратно проектувати розбиття системи на компоненти, прагнучи мінімізувати взаємодію між ними. Таким чином, основна складність при створенні мікроядерних операційних систем – необхідність дуже акуратного проектування.

Змішані системи

Всі розглянуті підходи до побудови операційних систем мають свої достоїнства і недоліки. В більшості випадків сучасні операційні системи використовують різні комбінації цих підходів. Так, наприклад, ядро операційної системи Linux є монолітною системою з елементами мікроядерної архітектури. При компіляції ядра можна вирішити динамічне завантаження і вивантаження дуже багатьох компонентів ядра – так званих модулів. У момент завантаження

модуля його код завантажується на рівні системи і зв'язується з рештою частини ядра. Усередині модуля можуть використовуватися будь-які функції, що експортуються ядром.

Іншим прикладом змішаного підходу може служити можливість запуску операційної системи з монолітним ядром під управлінням мікроядра. Такі влаштовані 4.4BSD і MkLinux, засновані на мікроядрі Mach. Мікроядро забезпечує управління віртуальною пам'яттю і роботу низькорівневих драйверів. Решта всіх функцій, зокрема взаємодії з прикладними програмами, здійснюється монолітним ядром. Даний підхід сформувався в результаті спроб використовувати переваги мікроядерної архітектури, зберігаючи по можливості добре відладжений код монолітного ядра.

Найтісніше елементи мікроядерної архітектури і елементи монолітного ядра переплетені в ядрі Windows NT. Хоча Windows NT часто називають мікроядерною операційною системою, це не зовсім так. Мікроядро NT дуже велике (більше 1 Мбайт), щоб носити приставку "мікро". Компоненти ядра Windows NT розташовуються в пам'яті, що витісняється, і взаємодіють один з одним шляхом передачі повідомлень, як і належить в мікроядерних операційних системах. В той же час всі компоненти ядра працюють в одному адресному просторі і активно використовують загальні структури даних, що властиве операційним системам з монолітним ядром. На думку фахівців Microsoft, причина проста: чисто мікроядерний дизайн комерційно невигідний, оскільки неефективний.

Таким чином, Windows NT можна з повним правом назвати гібридною операційною системою.

Класифікація ОС

Існує декілька схем класифікації операційних систем. Нижче приведена класифікація за деякими ознаками з погляду користувача.

Реалізація багатозадачності

По числу одночасно виконуваних завдань операційні системи можна розділити на два класи:

- багатозадачні (Unix, OS/2, Windows);
- однозадачні (наприклад, MS-DOS).

Однозадачні ОС в основному виконують функцію надання користувачеві віртуальної машини, роблячи простішим і зручнішим процес взаємодії користувача з комп'ютером. Однозадачні ОС включають засоби управління периферійними пристроями, засобу управління файлами, засобу спілкування з користувачем.

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

Витісняюча і невитісняюча багатозадачність.

Найважливішим ресурсом, що розділяється, є процесорний час. Спосіб розподілу процесорного часу між декількома процесами (або нитками), що одночасно існують в системі, багато в чому визначає специфіку ОС. Серед безлічі існуючих варіантів реалізації багатозадачності можна виділити дві групи алгоритмів:

невитісняюча багатозадачність (NetWare, Windows 3.x);

витісняюча багатозадачність (Windows NT, OS/2, UNIX).

Основною відмінністю між витісняючим і невитісняючим варіантами багатозадачності є ступінь централізації механізму планування процесів. У першому випадку механізм планування процесів цілком зосереджений в операційній системі, а в другому - розподілений між системою і прикладними програмами. При невитісняючій багатозадачності активний процес виконується до тих пір, поки він сам, за власною ініціативою, не віддасть управління операційній системі для того, щоб та вибрала з черги інший готовий до виконання процес. При витісняючій багатозадачності рішення про перемикання процесора з одного процесу на іншій ухвалюється операційною системою, а не самим активним процесом.

Підтримка многопользовательского режиму

По числу одночасно працюючих користувачів ОС можна розділити на:

- однокористувацькі (MS-DOS, Windows 3.x);
- багатокористувацькі (Windows NT, Unix).

Найбільш істотна відмінність між цими ОС полягає в наявності у багатокористувацьких систем механізмів захисту персональних даних кожного користувача.

Багатопроцесорна обробка

Аж до недавнього часу обчислювальні системи мали один центральний процесор. В результаті вимог до підвищення продуктивності з'явилися багатопроцесорні системи, що складаються з двох і більш за процесори загального призначення, що здійснюють паралельне виконання команд. Підтримка мультипроцесування є важливою властивістю ОС і приводить до ускладнення всіх алгоритмів управління ресурсами. Багатопроцесорна обробка реалізована в таких ОС, як Linux, Solaris, Windows NT, і ряду інших.

Багатопроцесорні ОС розділяють на симетричних і асиметричних. У симетричних ОС на кожному процесорі функціонує одне і те ж ядро, і завдання може бути виконана на будь-якому процесорі, тобто обробка повністю децентралізована. При цьому кожному з процесорів доступна вся пам'ять.

У асиметричних ОС процесори нерівноправні. Зазвичай існує головний процесор (master) і підлеглі (slave), завантаження і характер роботи яких визначає головний процесор.

Особливості областей використання

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

системи пакетної обробки (наприклад, ОС ЄС)

системи розділення часу (UNIX, VMS)

системи реального часу (QNX, RT/11).

Системи пакетної обробки призначалися для вирішення завдань в основному обчислювального характеру, що не вимагають швидкого отримання результатів. Головною метою і критерієм ефективності систем пакетної обробки є максимальна пропускна спроможність, тобто вирішення максимального числа завдань в одиницю часу. Для досягнення цієї мети в системах пакетної обробки використовуються наступна схема функціонування: на початку роботи формується пакет завдань, кожне завдання містить вимога до системних ресурсів; з цього пакету завдань формується мультипрограмна суміш, тобто безліч одночасно виконуваних завдань. Для одночасного виконання вибираються завдання, що пред'являють

вимоги, що відрізняються, до ресурсів, так, щоб забезпечувалося збалансоване завантаження всіх пристроїв обчислювальної машини; так, наприклад, в мультипрограмній суміші бажана одночасна присутність обчислювальних завдань і завдань з інтенсивним введенням-виводом. Таким чином, вибір нового завдання з пакету завдань залежить від внутрішньої ситуації, що складається в системі, тобто вибирається "вигідне" завдання. Отже, в таких ОС неможливо гарантувати виконання того або іншого завдання протягом певного періоду часу. У системах пакетної обробки перемикання процесора з виконання одного завдання на виконання інший відбувається тільки у випадку, якщо активне завдання саме відмовляється від процесора, наприклад, із-за необхідності виконати операцію введення-виводу. Тому одне завдання може надовго зайняти процесор, що робить неможливим виконання інтерактивних завдань. Таким чином, взаємодія користувача з обчислювальною машиною, на якій встановлена система пакетної обробки, зводиться до того, що він приносить завдання, віддає його диспетчерові-операторові, а в кінці дня після виконання всього пакету завдань отримує результат. Очевидно, що такий порядок знижує ефективність роботи користувача.

Системи розділення часу покликані виправити основний недолік систем пакетної обробки - ізоляцію користувача-програміста від процесу виконання його завдань. Кожному користувачеві системи розділення часу надається термінал, з якого він може вести діалог зі своєю програмою. Оскільки в системах розділення часу кожному завданню виділяється тільки квант процесорного часу, жодне завдання не займає процесор надовго, і час відповіді виявляється прийнятним. Якщо квант вибраний достатньо невеликим, то у всіх користувачів, що одночасно працюють на одній і тій же машині, складається враження, що кожен з них одноосібно використовує машину. Ясно, що системи розділення часу володіють меншою пропускною спроможністю, чим системи пакетної обробки, оскільки на виконання приймається кожне запущене користувачем завдання, а не та, яка "вигідна" системі, і, крім того, є накладні витрати обчислювальної потужності на частіше перемикання процесора із завдання на завдання. Критерієм ефективності систем розділення часу є не максимальна пропускна спроможність, а зручність і ефективність роботи користувача.

Системи реального часу застосовуються для управління різними технічними об'єктами, такими, наприклад, як верстат, супутник, наукова експериментальна установка або технологічними процесами, такими, як гальванічна лінія, доменний процес і тому подібне У всіх цих випадках існує гранично допустимий час, протягом якого має бути виконана та або інша програма, що управляє об'єктом, інакше може відбутися аварія: супутник вийде із зони видимості, експериментальні дані, що поступають з датчиків, будуть втрачені, товщина гальванічного покриття не відповідатиме нормі. Таким чином, критерієм ефективності для систем реального часу є їх здатність витримувати заздалегідь задані інтервали часу між запуском програми і отриманням результату (дії, що управляє). Цей час називається часом реакції системи, а відповідна властивість системи – реактивністю. Для цих систем мультипрограмна суміш є фіксованим набором заздалегідь розроблених програм, а вибір програми на виконання здійснюється виходячи з поточного стану об'єкту або відповідно до розкладу планових робіт.

Такі жорсткі обмеження позначаються на архітектурі систем реального часу, наприклад, в них може бути відсутньою віртуальна пам'ять, підтримка якої дає непередбачувані затримки у виконанні програм.

Деякі операційні системи можуть суміщати в собі властивості систем різних типів, наприклад, частина завдань може виконуватися в режимі пакетної обробки, а частина - в режимі реального часу або в режимі розділення часу. У таких випадках режим пакетної обробки часто називають фоновим режимом.

Висновок

Ми розглянули різні погляди на те, що таке операційна система; вивчили історію розвитку операційних систем; з'ясували, які функції зазвичай виконують операційні системи; нарешті, знялися на тому, які існують підходи до побудови операційних систем. Наступну лекцію ми присвятимо з'ясуванню поняття "процес" і питанням планування процесів.

Лекція 2.

Модуль 2. Управління процесами в операційних системах.

Починаючи з цієї лекції ми знайомитимемося з внутрішнім устроєм і механізмами дії операційних систем, розбираючи одну за іншою їх основні функції окремо і у взаємозв'язку. Фундаментальним поняттям для вивчення роботи операційних систем є поняття процесів як основних динамічних об'єктів, над якими системи виконують певні дії. Дана лекція присвячена опису таких об'єктів, їх станів і властивостей, їх уявленню в обчислювальних системах, а також операціям, які можуть проводитися над ними.

Поняття процесу

У першій лекції, пояснюючи поняття "Операційна система" і описуючи способи побудови операційних систем, ми часто застосовували слова "програма" і "завдання". Ми говорили: обчислювальна система виконує одну або декілька програм, операційна система планує завдання, програми можуть обмінюватися даними і так далі Ми використовували ці терміни в деякому загальновживаному, життєвому сенсі, припускаючи, що всі читачі однаково уявляють собі, що мається на увазі під ними у кожному конкретному випадку. При цьому одні і ті ж слова позначали і об'єкти в статичному стані, що не обробляються обчислювальною системою (наприклад, сукупність файлів на диску), і об'єкти в динамічному стані, що знаходяться в процесі виконання. Це було можливо, поки ми говорили про загальні властивості операційних систем, не вдаючись до подробиць їх внутрішнього устрою і поведінки, або про роботу обчислювальних систем першого та другого поколінь, які не могли обробляти більш за одну програму або одне завдання одночасно, по суті справи не маючи операційних систем. Але тепер ми починаємо знайомитися з деталями функціонування сучасних комп'ютерних систем, і нам доведеться уточнити термінологію.

Розглянемо наступний приклад. Два студенти запускають програму витягання квадратного кореня. Один хоче обчислити квадратний корінь з 4, а другий – з 1. З погляду студентів, запущена одна і та ж програма; з погляду комп'ютерної системи, їй доводиться займатися двома різними обчислювальними процесами, оскільки різні початкові дані приводять до різного набору обчислень. Отже, на рівні того, що відбувається усередині обчислювальної системи ми не можемо використовувати термін "програма" в призначеному для користувача сенсі слова.

Розглядаючи системи пакетної обробки, ми ввели поняття "завдання" як сукупність програми, набору команд мови управління завданнями, необхідних для її виконання, і вхідних даних. З погляду студентів, вони, підставивши різні початкові дані, сформували два різні завдання. Можливо, термін "завдання" підійде нам для опису внутрішнього функціонування комп'ютерних систем? Щоб з'ясувати це, давайте розглянемо інший приклад. Хай обидва студенти намагаються витягувати корінь квадратний з 1, тобто хай вони сформували ідентичні завдання, але завантажили їх в обчислювальну систему із зрушенням за часом. Тоді як одне з виконуваних завдань приступило до друку набутого значення і чекає закінчення операції введення-виводу, друге тільки починає виконуватися. Чи можна говорити про ідентичність завдань усередині обчислювальної системи в даний момент? Ні, оскільки стан процесу їх виконання різний. Отже, і слово "завдання" в призначеному для користувача сенсі не може застосовуватися для опису що відбувається в обчислювальній системі.

Це відбувається тому, що терміни "програма" і "завдання" призначені для опису статичних, неактивних об'єктів. Програма ж в процесі виконання є динамічним, активним об'єктом. По ходу її роботи комп'ютер обробляє різні команди і перетворить значення змінних. Для виконання програми операційна система повинна виділити певну кількість оперативної пам'яті, закріпити

за нею певні пристрої введення-виводу або файли (звідки повинні поступати вхідні дані і куди потрібно доставити отримані результати), тобто зарезервувати певні ресурси із загального числа ресурсів всієї обчислювальної системи. Їх кількість і конфігурація з часом можуть змінюватися. Для опису таких активних об'єктів усередині комп'ютерної системи замість термінів "програма" і "завдання" ми використовуватимемо новий термін – "процес".

У ряді навчальних посібників і монографій для простоти пропонується розглядати процес як абстракцію, що характеризує програму під час виконання. На наш погляд, ця рекомендація не зовсім коректна. Поняття процесу характеризує деяку сукупність набору команд, що виконуються, асоціюються з ним ресурсів (виділена для виконання пам'ять або адресний простір, стеки, використовувані файли і пристрої введення-виводу і т. д.) і теперішнього моменту його виконання (значення регістрів, програмного лічильника, стан стека і значення змінних), що знаходиться під управлінням операційної системи. Не існує взаємно-однозначної відповідності між процесами і програмами, оброблюваними обчислювальними системами. Як буде показано далі, в деяких операційних системах для роботи певних програм може організовуватися більш за один процес або один і той же процес може виконувати послідовно декілька різних програм. Більш того, навіть у разі обробки тільки одній програми в рамках одного процесу не можна вважати, що процесом є просто динамічний опис коди виконуваного файлу, даних і виділених для них ресурсів. Процес знаходиться під управлінням операційної системи, тому в нім може виконуватися частина коди її ядра (що не знаходиться у виконуваному файлі!), як у випадках, спеціально запланованих авторами програми (наприклад, при використанні системних викликів), так і в непередбачених ситуаціях (наприклад, при обробці зовнішніх переривань).

Стани процесу

При використанні такої абстракції все, що виконується в обчислювальних системах (не тільки програми користувачів, але і, можливо, певні частини операційних систем), організовано як набір процесів. Зрозуміло, що реально на однопроцесорній комп'ютерній системі в кожен момент часу може виконуватися тільки один процес. Для мультипрограмних обчислювальних систем псевдопаралельна обробка декількох процесів досягається за допомогою перемикання процесора з одного процесу на інший. Поки один процес виконується, останні чекають своєї черги.

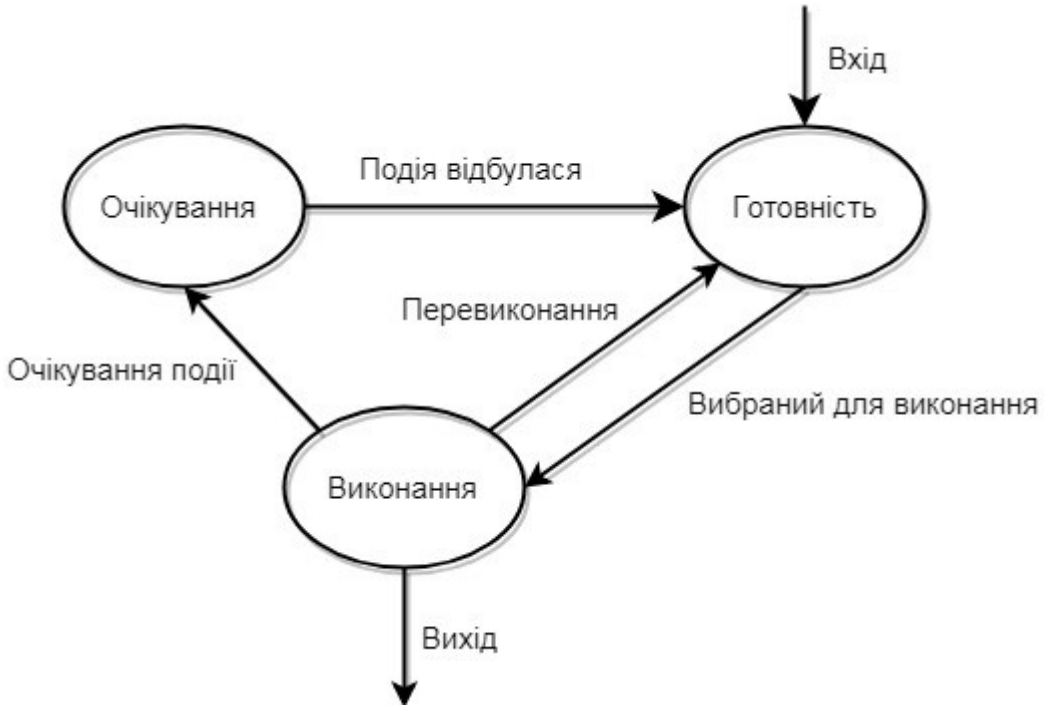
Як видимий, кожен процес може знаходитися як мінімум в двох станах: процес виконується і процес не виконується. Діаграма станів процесу в такій моделі зображена на мал. 2.1.

Мал. 2.1. Проста діаграма станів процесу

Процес, що знаходиться в змозі процес виконується, через деякий час може бути завершений операційною системою або припинений і знову переведений в стан процес не виконується. Припинення процесу відбувається по двох причинах: для його подальшої роботи було потрібно яку-небудь подію (наприклад, завершення операції введення-виводу) або закінчився часовий інтервал, відведений операційною системою для роботи даного процесу. Після цього операційна система по певному алгоритму вибирає для виконання один з процесів,

що знаходяться в змозі процес не виконується, і переводить його в стан процес виконується. Новий процес, що з'являється в системі, спочатку поміщається в стан процес не виконується.

Це дуже груба модель, вона не враховує, зокрема, те, що процес, вибраний для виконання, може все ще чекати події, із-за якої він був припинений, і реально до виконання не готовий. Для того, щоб уникнути такої ситуації, розіб'ємо стан процес не виконується на два нові стани: готовність і очікування (див. мал. 2.2).



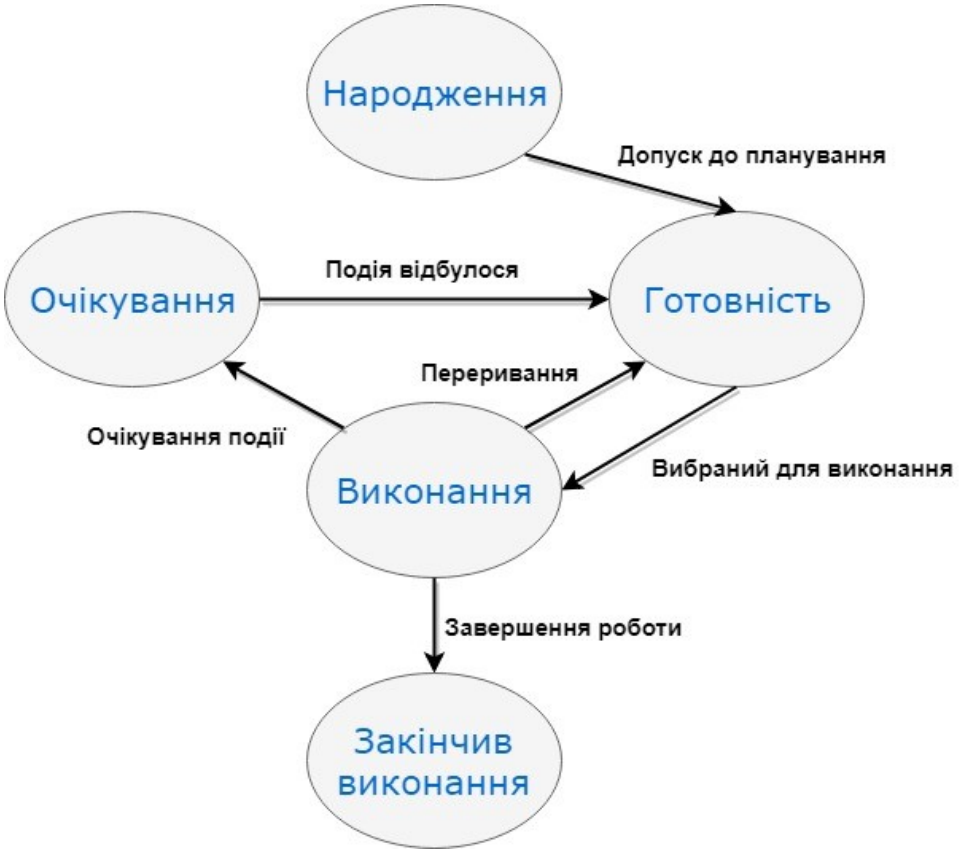
Мал. 2.2. Докладніша діаграма станів процесу

Всякий новий процес, що з'являється в системі, потрапляє в стан готовність. Операційна система, користуючись яким-небудь алгоритмом планування, вибирає один з готових процесів і переводить його в стан виконання. В змозі виконання відбувається безпосереднє виконання програмної коди процесу. Вийти з цього стану процес може по трьом причинам:

- операційна система припиняє його діяльність;
- він не може продовжувати свою роботу, поки не відбудеться деяка подія, і операційна система переводить його в стан очікування;
- в результаті виникнення переривання в обчислювальній системі (наприклад, переривання від таймера після закінчення передбаченого часу виконання) його повертають в стан готовність.

Із стану очікування процес потрапляє в стан готовність після того, як очікувана подія відбулася, і він знову може бути вибраний для виконання.

Наша нова модель добре описує поведінку процесів під час їх існування, але вона не акцентує уваги на появі процесу в системі і його зникненні. Для повноти картини нам необхідно ввести ще два стани процесів: народження і закінчив виконання (див. мал. 2.3).



Мал. 2.3. Діаграма станів процесу, прийнята в курсі

Тепер для появи в обчислювальній системі процес повинен пройти через стан народження. При народженні процес отримує в своє розпорядження адресний простір, в який завантажується програмний код процесу; йому виділяються стек і системні ресурси; встановлюється початкове значення програмного лічильника цього процесу і так далі процес, що Народився, переводиться в стан готовність. При завершенні своєї діяльності процес із стану виконання потрапляє в стан закінчив виконання.

У конкретних операційних системах стану процесу можуть бути ще більш деталізовані, можуть з'явитися деякі нові варіанти переходів з одного стану в інше. Так, наприклад, модель станів процесів для операційної системи Windows NT містить 7 різних станів, а для операційної системи Unix – 9. Проте так чи інакше, всі операційні системи підкоряються викладеній вище моделі.

Операції над процесами і пов'язані з ними поняття

Набор операцій

Процес не може перейти з одного стану в інше самостійно. Зміною стану процесів займається операційна система, здійснюючи операції над ними. Кількість таких операцій в нашій моделі поки збігається з кількістю стрілок на діаграмі станів. Зручно об'єднати їх в три пари:

- створення процесу – завершення процесу;
- припинення процесу (переклад із стану виконання в стан готовність) – запуск процесу (переклад із стану готовність в стан виконання);
- блокування процесу (переклад із стану виконання в стан очікування) – розблокування процесу (переклад із стану очікування в стан готовність).

Надалі, коли ми говоритимемо про алгоритми планування, в нашій моделі з'явиться ще одна операція, що не має парною: зміна пріоритету процесу.

Операції створення і завершення процесу є одноразовими, оскільки застосовуються до процесу не більше одного разу (деякі системні процеси при роботі обчислювальної системи не завершуються ніколи). Решта всіх операцій, пов'язаних із зміною стану процесів, будь то запуск або блокування, як правило, є багаторазовими. Розглянемо докладніше, як операційна система виконує операції над процесами.

Process Control Block і контекст процесу

Для того, щоб операційна система могла виконувати операції над процесами, кожен процес представляється в ній деякою структурою даних. Ця структура містить інформацію, специфічну для даного процесу:

- стан, в якому знаходиться процес;
- програмний лічильник процесу або, іншими словами, адреса команди, яка має бути виконана для нього наступною;
- вміст регістрів процесора;
- дані, необхідні для планування використання процесора і управління пам'яттю (пріоритет процесу, розмір і розташування адресного простору і т. д.);
- облікові дані (ідентифікаційний номер процесу, який користувач ініціював його роботу, загальний час використання процесора даним процесом і т. д.);
- відомості про пристрої введення-виводу, пов'язані з процесом (наприклад, які пристрої закріплені за процесом, таблицю відкритих файлів).

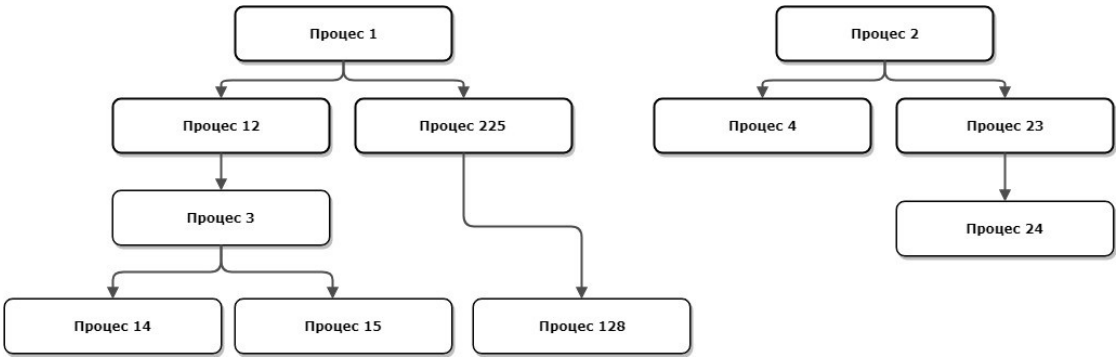
Її склад і будова залежать, звичайно, від конкретної операційної системи. У багатьох операційних системах інформація, що характеризує процес, зберігається не в одній, а в декількох зв'язаних структурах даних. Ці структури можуть мати різні найменування, містити додаткову інформацію або, навпаки, лише частину описаної інформації. Для нас це не має значення. Для нас важливо лише те, що для будь-якого процесу, що знаходиться в обчислювальній системі, вся інформація, необхідна для здійснення операцій над ним, доступна операційній системі. Для простоти викладу вважатимемо, що вона зберігається в одній структурі даних. Ми називатимемо її PCB (Process Control Block) або блоком управління процесом. Блок управління процесом є моделлю процесу для операційної системи. Будь-яка операція, вироблювана операційною системою над процесом, викликає певні зміни в PCB. В рамках прийнятої моделі станів процесів вміст PCB між операціями залишається постійним.

Інформацію, для зберігання якої призначений блок управління процесом, зручно для подальшого викладу розділити на дві частини. Вміст всіх регістрів процесора (включаючи значення програмного лічильника) називатимемо регістровим контекстом процесу, а все останнє – системним контекстом процесу. Знання регістрового і системного контекстів процесу досить для того, щоб управляти його роботою в операційній системі, здійснюючи над ним операції. Проте цього недостатньо для того, щоб повністю охарактеризувати процес. Операційну систему не цікавить, якими саме обчисленнями займається процес, тобто який код і які дані знаходяться в його адресному просторі. З погляду користувача, навпаки, найбільший інтерес представляє вміст адресного простору процесу, можливо, разом з регістровим контекстом що визначає послідовність перетворення даних і отримані результати. Код і дані, що знаходяться в адресному просторі процесу, називатимемо його призначеним для користувача контекстом. Сукупність регістрового, системного і призначеного для користувача

контекстів процесу скорочено прийнято називати просто контекстом процесу. У будь-який момент часу процес повністю характеризується своїм контекстом.

Одноразові операції

Складний життєвий шлях процесу в комп'ютері починається з його народження. Будь-яка операційна система, що підтримує концепцію процесів, повинна володіти засобами для їх створення. У дуже простих системах (наприклад, в системах, спроектованих для роботи тільки одного конкретного застосування) всі процеси можуть бути породжені на етапі старту системи. Складніші операційні системи створюють процеси динамічно, в міру необхідності. Ініціатором народження нового процесу після старту операційної системи може виступити або процес користувача, що зробив спеціальний системний виклик, або сама операційна система, тобто, зрештою, теж деякий процес. Процес, що ініціював створення нового процесу, прийнято називати процесом-батьком (parent process), а знов створений процес – процесом-дитиною (child process). Процеси-діти можуть у свою чергу породжувати нових дітей і т. д., утворюючи, в загальному випадку, усередині системи набір генеалогічних дерев процесів – генеалогічний ліс. Приклад генеалогічного лісу зображений на малюнку 2.4. Слід зазначити, що всі призначені для користувача процеси разом з деякими процесами операційної системи належать одному і тому ж дереву лісу. У ряді обчислювальних систем ліс взагалі вироджується в одне таке дерево.



Мал. 2.4. Спрощений генеалогічний ліс процесів. Стрілка означає відношення батько–дитина

При народженні процесу система заводить новий PCB із станом процесу народження і починає його заповнювати. Новий процес отримує власний унікальний ідентифікаційний номер. Оскільки для зберігання ідентифікаційного номера процесу в операційній системі відводиться обмежена кількість бітів, для дотримання унікальності номерів кількість одночасно присутніх в ній процесів має бути обмежене. Після завершення якого-небудь процесу його ідентифікаційний номер, що звільнився, може бути повторно використаний для іншого процесу.

Зазвичай для виконання своїх функцій процес-дитина вимагає певних ресурсів: пам'яті, файлів, пристроїв введення-виводу і так далі Існує два підходи до їх виділення. Новий процес може отримати в своє розпорядження деяку частину батьківських ресурсів, можливо розділяючи з процесом-батьком і іншими процесами-дітьми права на них, або може отримати свої ресурси безпосередньо від операційної системи. Інформація про виділені ресурси заноситься в PCB.

Після наділу процесу-дитини ресурсами необхідно занести в його адресний простір програмний код, значення даних, встановити програмний лічильник. Тут також можливі два рішення. У першому випадку процес-дитина стає дублікатом процесу-батька по регістровому і призначеному для користувача контекстах, при цьому повинен існувати спосіб визначення, хто для кого з процесів-двійників є батьком. У другому випадку процес-дитина завантажується новою програмою з якого-небудь файлу. Операційна система Unix вирішує породження процесу тільки першим способом; для запуску нової програми необхідно спочатку створити копію

процесу-батька, а потім процес-дитина повинен замінити свій призначений для користувача контекст за допомогою спеціального системного виклику. Операційна система VAX/VMS допускає тільки друге рішення. У Windows NT можливі обидва варіанти (у різних API).

Породження нового процесу як дубліката процесу-батька приводить до можливості існування програм (тобто виконуваних файлів), для роботи яких організовується більш за один процес. Можливість заміни призначеного для користувача контексту процесу по ходу його роботи (тобто завантаження для виконання нової програми) приводить до того, що в рамках одного і того ж процесу може послідовно виконуватися декілька різних програм.

Після того, як процес наділений змістом, в PCB дописується інформація, що залишилася, і стан нового процесу змінюється на готовність. Залишилося сказати декілька слів про те, як поводяться процеси-батьки після народження процесів-дітей. Процес-батько може продовжувати своє виконання одночасно з виконанням процесу-дитини, а може чекати завершення роботи деяких або всіх своїх "дітей".

Ми детально не зупинятимемося на причинах, які можуть привести до завершення життєвого циклу процесу. Після того, як процес завершив свою роботу, операційна система переводить його в стан закінчив виконання і звільняє всі асоційовані з ним ресурси, роблячи відповідні записи в блоці управління процесом. При цьому сам PCB не знищується, а залишається в системі ще якийсь час. Це пов'язано з тим, що процес-батько після завершення процесу-дитини може запитати операційну систему про причину "смерті" породженого ним процесу і/або статистичну інформацію про його роботу. Подібна інформація зберігається в PCB відпрацьованого процесу до запиту процесу-батька або до кінця його діяльності, після чого всі сліди процесу, що завершився, остаточно зникають з системи. У операційній системі Unix процеси, що знаходяться в змозі закінчив виконання, прийнято називати процесами-зомбі.

Слід відмітити, що у ряді операційних систем (наприклад, в VAX/VMS) загибель процесу-батька приводить до завершення роботи всіх його "дітей". У інших операційних системах (наприклад, в Unix) процеси-діти продовжують своє існування і після закінчення роботи процесу-батька. При цьому виникає необхідність зміни інформації в PCB процесів-дітей про породжувач їх процесі для того, щоб генеалогічний ліс процесів залишався цілісним. Розглянемо наступний приклад. Хай процес з номером 2515 був породжений процесом з номером 2001 і після завершення його роботи залишається в обчислювальній системі необмежено довго. Тоді не виключено, що номер 2001 буде використаний операційною системою повторно для зовсім іншого процесу. Якщо не змінити інформацію про процес-батько для процесу 2515, то генеалогічний ліс процесів виявиться некоректним – процес 2515 вважатиме за свого батька новий процес 2001, а процес 2001 відхрещуватиметься від нежданого нащадка. Як правило, що "усиротіли" процеси "усиновляються" одним з системних процесів, який породжується при старті операційної системи і функціонує весь час, поки вона працює.

Багаторазові операції

Одноразові операції приводять до зміни кількості процесів, що знаходяться під управлінням операційної системи, і завжди пов'язані з виділенням або звільненням певних ресурсів. Багаторазові операції, навпаки, не приводять до зміни кількості процесів в операційній системі і не зобов'язані бути пов'язаними з виділенням або звільненням ресурсів.

У цьому розділі ми коротко опишемо дії, які проводить операційна система при виконанні багаторазових операцій над процесами. Детальніше ці дії будуть розглянуті далі у відповідних лекціях.

Запуск процесу. З числа процесів, що знаходяться в змозі готовність, операційна система вибирає один процес для подальшого виконання. Критерії і алгоритми такого вибору будуть детально розглянуті в лекції 3 – "Планування процесів". Для вибраного процесу операційна система забезпечує наявність в оперативній пам'яті інформації, необхідної для його подальшого виконання. То, як вона це робить, буде в деталях описано в лекціях 8-10. Далі стан процесу змінюється на виконання, відновлюються значення регістрів для даного процесу і управління передається команді, на яку указує лічильник команд процесу. Всі дані, необхідні для відновлення контексту, витягуються з PCB процесу, над яким здійснюється операція.

Припинення процесу. Робота процесу, що знаходиться в змозі виконання, припиняється в результаті якого-небудь переривання. Процесор автоматично зберігає лічильник команд і, можливо, один або декілька регістрів в стеку виконуваного процесу, а потім передає управління за спеціальною адресою обробки даного переривання. На цьому діяльність hardware по обробці переривання завершується. За вказаною адресою зазвичай розташовується одна з частин операційної системи. Вона зберігає динамічну частину системного і регістрового контекстів процесу в його PCB, переводить процес в стан готовність і приступає до обробки переривання, тобто до виконання певних дій, пов'язаних з виниклим перериванням.

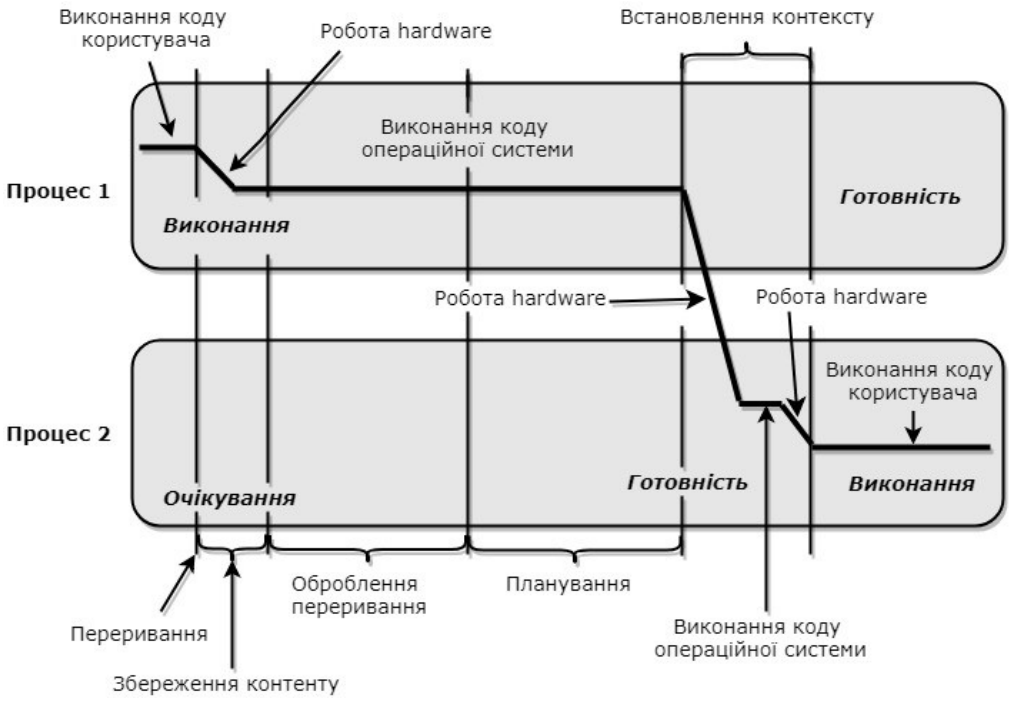
Блокування процесу. Процес блокується, коли він не може продовжувати роботу, не дочекавшись виникнення якої-небудь події в обчислювальній системі. Для цього він звертається до операційної системи за допомогою певного системного виклику. Операційна система обробляє системний виклик (ініціалізував операцію введення-виводу, додає процес в чергу процесів, що чекають звільнення пристрою або виникнення події, і т. д.) і, при необхідності зберігши потрібну частину контексту процесу в його PCB, переводить процес із стану виконання в стан очікування. Докладніше ця операція розглядатиметься в лекції 13.

Розблокування процесу. Після виникнення в системі якої-небудь події операційній системі потрібно точно визначити, яке саме подія відбулася. Потім операційна система перевіряє, чи знаходився деякий процес в змозі очікування для даної події, і якщо знаходився, переводить його в стан готовність, виконуючи необхідні дії, пов'язані з настанням події (ініціалізація операції введення-виводу для чергового чекаючого процесу і т. п.). Ця операція, як і операція блокування, буде детально описана в лекції 13.

Перемикання контексту

До цих пір ми розглядали операції над процесами ізольовано, незалежно один від одного. Насправді ж діяльність мультипрограмної операційної системи складається з ланцюжків операцій, що виконуються над різними процесами, і супроводиться перемиканням процесора з одного процесу на іншій.

Давайте для прикладу спрощено розглянемо, як в реальності може протікати операція розблокування процесу, чекаючого введення-виводу (див. мал. 2.5). При виконання процесором деякого процесу (на малюнку – процес 1) виникає переривання від пристрою введення-виводу, що сигналізує про закінчення операцій на пристрої. Над процесом, що виконується, проводиться операція припинення. Далі операційна система розблоковує процес, що ініціював запит на уведення-виведення (на малюнку – процес 2) і здійснює запуск припиненого або нового процесу, вибраного при виконанні планування (на малюнку був вибраний розблокований процес). Як ми бачимо, в результаті обробки інформації про закінчення операції введення-виводу можлива зміна процесу, що знаходиться в змозі виконання.



Мал. 2.5. Виконання операції розблокування процесу. Використання терміну "код користувача" не обмежує спільності малюнка тільки призначеними для користувача процесами

Для коректного перемикання процесора з одного процесу на іншій необхідно зберегти контекст процесу, що виконувався, і відновити контекст процесу, на який буде перемкнутий процесор. Така процедура збереження/відновлення працездатності процесів називається перемиканням контексту. Час, витрачений на перемикання контексту, не використовується обчислювальною системою для здійснення корисної роботи і є накладними витратами, що знижують продуктивність системи. Воно міняється від машини до машини і зазвичай коливається в діапазоні від 1 до 1000 мікросекунд. Істотно скоротити накладні витрати в сучасних операційних системах дозволяє розширена модель процесів, що включає поняття threads of execution (нитки виконання або просто нитки).

Висновок

Поняття процесу характеризує деяку сукупність набору команд, що виконуються, асоціюються з ним ресурсів і теперішнього моменту його виконання, що знаходиться під управлінням операційної системи. У будь-який момент процес повністю описується своїм контекстом, що складається з регістрової, системної і призначеної для користувача частин. У операційній системі процеси представляються певною структурою даних – PCB, що відображає зміст регістрового і системного контекстів. Процеси можуть знаходитися в п'яти основних станах: народження, готовність, виконання, очікування, закінчив виконання. Із стану в стан процес переводиться операційною системою в результаті виконання над ним операцій. Операційна система може виконувати над процесами наступні операції: створення процесу, завершення процесу, припинення процесу, запуск процесу, блокування процесу, розблокування процесу, зміна пріоритету процесу. Між операціями вміст PCB не змінюється. Діяльність мультипрограмної операційної системи складається з ланцюжків перерахованих операцій, що виконуються над різними процесами, і супроводиться процедурами збереження/відновлення працездатності процесів, тобто перемиканням контексту. Перемикання контексту не має відношення до корисної роботи, що виконується процесами, і час, витрачений на нього, скорочує корисний час роботи процесора.

Лекція 3. Управління процесами в операційних системах.

Планування процесів

Рівні планування

У першій лекції, розглядаючи еволюцію комп'ютерних систем, ми говорили про два види планування в обчислювальних системах: плануванні завдань і плануванні використання процесора. Планування завдань з'явилося в пакетних системах після того, як для зберігання сформованих пакетів завдань почали використовуватися магнітні диски. Магнітні диски, будучи пристроями прямого доступу, дозволяють завантажувати завдання в комп'ютер в довільному порядку, а не тільки в тому, в якому вони були записані на диск. Змінюючи порядок завантаження завдань в обчислювальну систему, можна підвищити ефективність її використання. Процедуру вибору чергового завдання для завантаження в машину, тобто для породження відповідного процесу, ми і назвали плануванням завдань. Планування використання процесора вперше виникає в мультипрограмних обчислювальних системах, де в змозі готовність можуть одночасно знаходитися декілька процесів. Саме для процедури вибору з них одного процесу, який отримає процесор в своє розпорядження, тобто буде переведений в стан виконання, ми використовували цю словосполуку. Тепер, познайомившись з концепцією процесів в обчислювальних системах, обидва види планування ми розглядатимемо як різні рівні планування процесів.

Планування завдань використовується як довгострокове планування процесів. Воно відповідає за породження нових процесів в системі, визначаючи її ступінь мультипрограмування, тобто кількість процесів, що одночасно знаходяться в ній. Якщо ступінь мультипрограмування системи підтримується постійним, тобто середня кількість процесів в комп'ютері не міняється, то нові процеси можуть з'являтися тільки після завершення раніше завантажених. Тому довгострокове планування здійснюється достатньо рідко, між появою нових процесів можуть проходити хвилини і навіть десятки хвилин. Рішення про вибір для запуску того або іншого процесу робить вплив на функціонування обчислювальної системи впродовж достатнього тривалого часу. Звідси і назва цього рівня планування – довгострокове. У деяких операційних системах довгострокове планування зведене до мінімуму або відсутнє зовсім. Так, наприклад, в багатьох інтерактивних системах розділення часу породження процесу відбувається відразу після появи відповідного запиту. Підтримка розумного ступеня мультипрограмування здійснюється за рахунок обмеження кількості користувачів, які можуть працювати в системі, і особливостей людської психології. Якщо між натисненням на клавішу і появою символу на екрані проходить 20–30 секунд, то багато користувачів вважатимуть за краще припинити роботу і продовжити її, коли система буде менш завантажена.

Планування використання процесора застосовується як короткострокове планування процесів. Воно проводиться, наприклад, при зверненні процесу, що виконується, до пристроїв введення-виводу або просто після закінчення певного інтервалу часу. Тому короткострокове планування здійснюється, як правило, не рідше за один раз в 100 мілісекунд. Вибір нового процесу для виконання робить вплив на функціонування системи до настання чергової аналогічної події, тобто протягом короткого проміжку часу, чим і обумовлена назва цього рівня планування – короткострокове.

У деяких обчислювальних системах буває вигідно для підвищення продуктивності тимчасово видалити який-небудь процес, що частково виконався, з оперативної пам'яті на диск, а пізніше повернути його назад для подальшого виконання. Така процедура в англомовній літературі отримала назву swapping, що можна перекласти російською мовою як "перекачування", хоча в спеціальній літературі воно уживається без перекладу – свопінг. Коли і яким з процесів потрібно перекачати на диск і повернути назад, вирішується додатковим проміжним рівнем планування процесів – середньостроковим.

Критерії планування і вимоги до алгоритмів

Для кожного рівня планування процесів можна запропонувати багато різних алгоритмів. Вибір конкретного алгоритму визначається класом завдань, що вирішуються обчислювальною системою, і цілями, яких ми хочемо досягти, використовуючи планування. До таких цілей можна віднести наступні:

- Справедливість – гарантувати кожному завданню або процесу певну частину часу використання процесора в комп'ютерній системі, прагнучи не допустити виникнення ситуації, коли процес одного користувача постійно займає процесор, тоді як процес іншого користувача фактично не починав виконуватися.
- Ефективність – постаратися зайняти процесор на все 100% робочого часу, не дозволяючи йому простоювати в очікуванні процесів, готових до виконання. У реальних обчислювальних системах завантаження процесора коливається від 40 до 90%.
- Скорочення повного часу виконання (turnaround time) – забезпечити мінімальний час між стартом процесу або постановкою завдання в чергу для завантаження і його завершенням.
- Скорочення часу очікування (waiting time) – скоротити час, який проводять процеси в змозі готовність і завдання в черзі для завантаження.
- Скорочення часу відгуку (response time) – мінімізувати час, який потрібний процесу в інтерактивних системах для відповіді на запит користувача.

Незалежно від поставлених цілей планування бажано також, щоб алгоритми володіли наступними властивостями.

- Були передбаченими. Одне і те ж завдання повинне виконуватися приблизно за один і той же час. Застосування алгоритму планування не повинне приводити, наприклад, до витягання кореня квадратного з 4 за соті долі секунди при одному запуску і за декілька діб – при другому запуску.
- Були пов'язані з мінімальними накладними витратами. Якщо на кожних 100 мілісекунд, виділені процесу для використання процесора, доводитиметься 200 мілісекунд на визначення того, який саме процес отримає процесор в своє розпорядження, і на перемикання контексту, то такий алгоритм, очевидно, застосовувати не варто.
- Рівномірно завантажували ресурси обчислювальної системи, віддаючи перевагу тим процесам, які займатимуть маловикористовувані ресурси.
- Володіли масштабованістю, тобто не відразу втрачали працездатність при збільшенні навантаження. Наприклад, зростання кількості процесів в системі в два рази не повинне приводити до збільшення повного часу виконання процесів на порядок.

Багато хто з приведених вище цілей і властивостей є суперечливим. Покращуючи роботу алгоритму з погляду одного критерію, ми погіршуємо її з погляду іншого. Пристосовувавши алгоритм під один клас завдань, ми тим самим дискримінуємо завдання іншого класу. "У один віз упрягти не можна коня і трепетну лань". Нічого не поробиш. Таке життя.

Параметри планування

Для здійснення поставлених цілей розумні алгоритми планування повинні спиратися на які-небудь характеристики процесів в системі, завдань в черзі на завантаження, стани самої обчислювальної системи, іншими словами, на параметри планування. У цьому розділі ми опишемо ряд таких параметрів, не претендуючи на повноту викладу.

Всі параметри планування можна розбити на дві великі групи: статичні параметри і динамічні параметри. Статичні параметри не змінюються в ході функціонування обчислювальної системи, динамічні ж, навпаки, схильні до постійних змін.

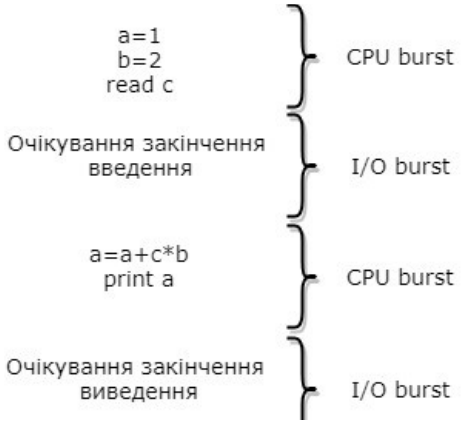
До статичних параметрів обчислювальної системи можна віднести граничні значення її ресурсів (розмір оперативної пам'яті, максимальна кількість пам'яті на диску для здійснення свопінгу, кількість підключених пристроїв введення-виводу і т. п.). Динамічні параметри системи описують кількість вільних ресурсів на даний момент.

До статичних параметрів процесів відносяться характеристики, як правило властиві завданням вже на етапі завантаження.

- Яким користувачем запущений процес або сформовано завдання.
- Наскільки важливим є поставлене завдання, тобто який пріоритет її виконання.
- Скільки процесорного часу запитано користувачем для вирішення завдання.
- Яке співвідношення процесорного часу і часу, необхідного для здійснення операцій введення-виводу.
- Які ресурси обчислювальної системи (оперативна пам'ять, пристрої введення-виводу, спеціальні бібліотеки і системні програми і т. д.) і в якій кількості необхідні завданню.

Алгоритми довгострокового планування використовують в своїй роботі статичні і динамічні параметри обчислювальної системи і статичні параметри процесів (динамічні параметри процесів на етапі завантаження завдань ще не відомі). Алгоритми короткострокового і середньострокового планування додатково враховують і динамічні характеристики процесів. Для середньострокового планування як такі характеристики може використовуватися наступна інформація:

- скільки часу пройшло з моменту вивантаження процесу на диск або його завантаження в оперативну пам'ять;
- скільки оперативної пам'яті займає процес;
- скільки процесорного часу вже надано процесу.



Мал. 3.1. Фрагмент діяльності процесу з виділенням проміжків безперервного використання процесора і очікування введення-виводу

Для короткострокового планування нам знадобиться ввести ще два динамічні параметри. Діяльність будь-якого процесу можна представити як послідовність циклів використання процесора і очікування завершення операцій введення-виводу. Проміжок часу безперервного використання процесора носить назва CPU burst, а проміжок часу безперервного очікування введення-виводу – I/O burst. На мал. 3.1 показаний фрагмент діяльності деякого процесу на псевдомові програмування з виділенням вказаних проміжків. Скорочено ми використовуватимемо терміни CPU burst і I/O burst без перекладу. Значення тривалості останніх і чергових CPU burst і I/O burst є важливими динамічними параметрами процесу.

Витісняюче і невитісняюче планування

Процес планування здійснюється частиною операційної системи, званої планувальником. Планувальник може ухвалювати рішення про вибір для виконання нового процесу з числа тих, що знаходяться в змозі готовність в наступних чотирьох випадках.

1. Коли процес переводиться із стану виконання в стан закінчив виконання.
2. Коли процес переводиться із стану виконання в стан очікування.
3. Коли процес переводиться із стану виконання в стан готовність (наприклад, після переривання від таймера).
4. Коли процес переводиться із стану очікування в стан готовність (завершилася операція введення-виводу або відбулася інша подія). Детально процедура такого перекладу розглядалася в лекції 2 (розділ "Перемикання контексту"), де ми показали, чому при цьому виникає можливість зміни процесу, що знаходиться в змозі виконання.

У випадках 1 і 2 процес, що знаходився в змозі виконання, не може далі виконуватися, і операційна система вимушена здійснювати планування вибираючи новий процес для виконання. У випадках 3 і 4 планування може як проводитися, так і не проводитися, планувальник не вимушений обов'язково ухвалювати рішення про вибір процесу для виконання, процес, що знаходився в змозі виконання може просто продовжити свою роботу. Якщо в операційній системі планування здійснюється тільки у вимушених ситуаціях, говорять, що має місце невитісняюче (nonpreemptive) планування. Якщо планувальник приймає і вимушені, і невимушені рішення, говорять про витісняюче (preemptive) планування. Термін "витісняюче планування" виник тому, що процес, що виконується, окрім своєї волі може бути витиснений із стану виконання іншим процесом.

Невитісняюче планування використовується, наприклад, в MS Windows 3.1 і ОС Apple Macintosh. При такому режимі планування процес займає стільки процесорного часу, скільки йому необхідно. При цьому перемикання процесів виникає тільки процесу, що за бажання самого виконується, передати управління (для очікування завершення операції введення-виводу або після закінчення роботи). Цей метод планування відносно просто реалізовується і достатньо ефективний, оскільки дозволяє виділити велику частину процесорного часу для роботи самих процесів і до мінімуму скоротити витрати на перемикання контексту. Проте при невитісняючому плануванні виникає проблема можливості повного захоплення процесора одним процесом, який унаслідок яких-небудь причин (наприклад, із-за помилки в програмі) зациклюється і не може передати управління іншому процесу. У такій ситуації рятує тільки перезавантаження всієї обчислювальної системи.

Витісняюче планування зазвичай використовується в системах розділення часу. У цьому режимі планування процес може бути припинений у будь-який момент виконань. Операційна система встановлює спеціальний таймер для генерації сигналу переривання після закінчення деякого інтервалу часу – кванта. Після переривання процесор передається в розпорядження наступного процесу. Тимчасові переривання допомагають гарантувати прийнятний час відгуку

процесів для користувачів, що працюють в діалоговому режимі, і запобігають "зависанню" комп'ютерної системи із-за зациклення якої-небудь програми.

Алгоритми планування

Існує достатньо великий набір різноманітних алгоритмів планування, які призначені для досягнення різних цілей і ефективні для різних класів завдань. Багато хто з них може використовуватися на декількох рівнях планування. У цьому розділі ми розглянемо деякі найбільш споживані алгоритми стосовно процесу короткочасного планування.

First-Come, First-Served (FCFS)

Простим алгоритмом планування є алгоритм, який прийнято позначати абревіатурою FCFS по перших буквах його англійської назви, – First-Come, First-Served (першим прийшов, першим обслужений). Уявимо собі, що процеси, що знаходяться в змозі готовність, збудовані в чергу. Коли процес переходить в стан готовність, він, а точніше, посилання на його PCB поміщається в кінець цієї черги. Вибір нового процесу для виконання здійснюється з початку черги з видаленням звідти посилання на його PCB. Черга подібного типу має в програмуванні спеціальне найменування – FIFO1), скорочення від First In, First Out (першим увійшов, першим вийшов).

Такий алгоритм вибору процесу здійснює невитісняюче планування. Процес, що отримав в своє розпорядження процесор, займає його до закінчення поточного CPU burst. Після цього для виконання вибирається новий процес з початку черги.

Таблиця 3.1.

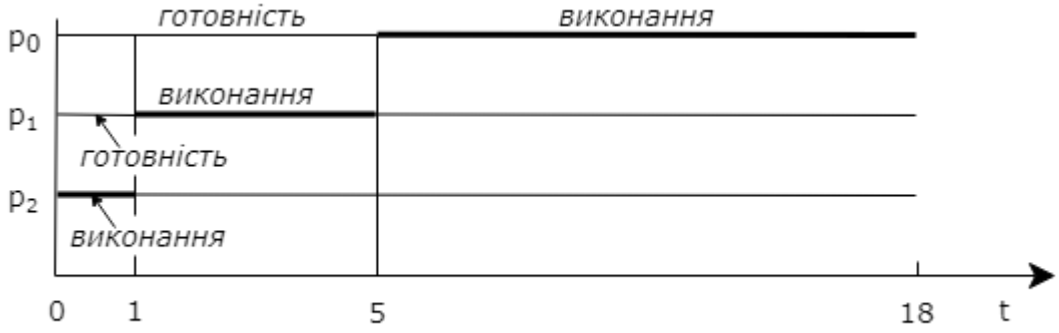
Процес	p0	p1	p2
Тривалість чергового CPU burst	13	4	1

Перевагою алгоритму FCFS є легкість його реалізації, але в той же час він має і багато недоліків. Розглянемо наступний приклад. Хай в змозі готовність знаходяться три процеси p0, p1 і p2, для яких відомі часи їх чергових CPU burst. Ці часи приведені в таблиця 3.1 в деяких умовних одиницях. Для простоти вважатимемо, що вся діяльність процесів обмежується використанням тільки одного проміжку CPU burst, що процеси не здійснюють операцій введення-виводу і що час перемикання контексту так мало, що їм можна нехтувати.

Якщо процеси розташовані в черзі процесів, готових до виконання, в порядку p0, p1, p2, то картина їх виконання виглядає так, як показано на мал. 3.2. Першим для виконання вибирається процес p0, який отримує процесор на весь час свого CPU burst, тобто на 13 одиниць часу. Після його закінчення в стан виконання переводиться процес p1, він займає процесор на 4 одиниці часу. І, нарешті, можливість працювати отримує процес p2. Час очікування для процесу p0 складає 0 одиниць часу, для процесу p1 – 13 одиниць, для процесу p2 – 13 + 4 = 17 одиниць. Таким чином, середній час очікування в цьому випадку – (0 + 13 + 17) / 3 = 10 одиниць часу. Повний час виконання для процесу p0 складає 13 одиниць часу, для процесу p1 – 13 + 4 = 17 одиниць, для процесу p2 – 13 + 4 + 1 = 18 одиниць. Середній повний час виконання виявляється рівним (13 + 17 + 18) /3 = 16 одиницям часу.

Мал. 3.2. Виконання процесів при порядку p0,p1,p2

Якщо ті ж самі процеси розташовані в порядку p2, p1, p0, то картина їх виконання відповідатиме мал. 3.3. Час очікування для процесу p0 дорівнює 5 одиницям часу, для процесу p1 – 1 одиниці, для процесу p2 – 0 одиниць. Середній час очікування складе (5 + 1 + 0) /3 = 2 одиниці часу. Це в 5 (!) разів менше, ніж у попередньому випадку. Повний час виконання для процесу p0 виходить рівним 18 одиницям часу, для процесу p1 – 5 одиницям, для процесу p2 – 1 одиниці. Середній повний час виконання складає (18 + 5 + 1) /3 = 8 одиниць часу, що майже в 2 рази менше, ніж при першій розстановці процесів.

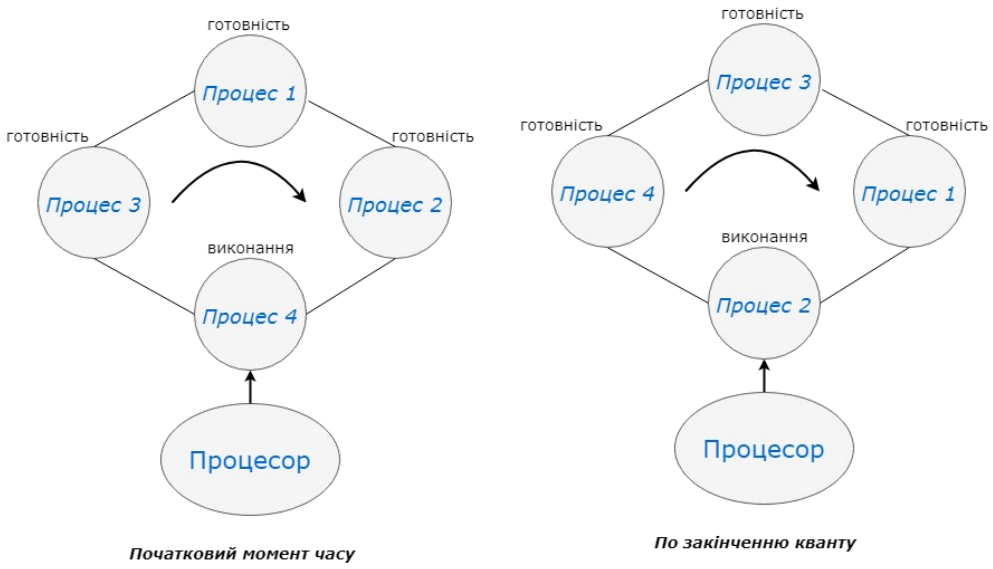


Мал. 3.3. Виконання процесів при порядку p2, p1, p0

Як ми бачимо, середній час очікування і середній повний час виконання для цього алгоритму істотно залежать від порядку розташування процесів в черзі. Якщо у нас є процес з тривалим CPU burst, то короткі процеси, що перейшли в стан готовність після тривалого процесу, будуть дуже довго чекати почала виконання. Тому алгоритм FCFS практично непридатний для систем розділення часу – дуже великим виходить середній час відгуку в інтерактивних процесах.

Round Robin (RR)

Модифікацією алгоритму FCFS є алгоритм, що отримав назву Round Robin (Round Robin – це вид дитячої каруселі в США) або скорочено RR. По суті справи, це той же самий алгоритм, тільки реалізований в режимі витісняючого планування. Можна уявити собі всю безліч готових процесів організованою циклічно – процеси сидять на каруселі. Карусель обертається так, що кожен процес знаходиться біля процесора невеликий фіксований квант часу, звичайні 10 – 100 мілісекунд (див. мал. 3.4). Поки процес знаходиться поряд з процесором, він отримує процесор в своє розпорядження і може виконуватися.



Мал. 3.4. Процеси на каруселі

Реалізується такий алгоритм так само, як і попередній, за допомогою організації процесів, що знаходяться в змозі готовність, в чергу FIFO. Планувальник вибирає для чергового виконання процес, розташований на початку черги, і встановлює таймер для генерації переривання після закінчення певного кванта часу. При виконанні процесу можливі два варіанти.

- Час безперервного використання процесора, необхідне процесу (залишок поточного CPU burst), менше або дорівнює тривалості кванта часу. Тоді процес по своїй волі звільняє процесор до закінчення кванта часу, на виконання поступає новий процес з початку черги, і таймер починає відлік кванта наново.
- Тривалість залишку поточного CPU burst процесу більша, ніж квант часу. Тоді після закінчення цього кванта процес уривається таймером і поміщається в кінець черги процесів, готових до виконання, а процесор виділяється для використання процесу, що знаходиться на її початку.

Розглянемо попередній приклад з порядком процесів p0, p1, p2 і величиною кванта часу рівної 4. Виконання цих процесів ілюструється таблиця 3.2. Позначення "І" використовується в ній для процесу, що знаходиться в змозі виконання, позначення "Г" – для процесів в змозі готовність, порожні осередки відповідають процесам, що завершилися. Стани процесів показані впродовж відповідної одиниці часу, тобто колонка з номером 1 відповідає проміжку часу від 0 до 1.

Таблиця 3.2.

Час	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	І	І	І	І	Грам	Грам	Грам	Грам	Грам	І	І	І	І	І	І	І	І	І
p1	Грам	Грам	Грам	Грам	І	І	І	І										
p2	Грам	Грам	Грам	Грам	Грам	Грам	Грам	Грам	І									

Першим для виконання вибирається процес p0. Тривалість його CPU burst більше, ніж величина кванта часу, і тому процес виконується до закінчення кванта, тобто протягом 4 одиниць часу. Після цього він поміщається в кінець черги готових до виконання процесів, яка набирає вигляду p1, p2, p0. Наступним починає виконуватися процес p1. Час його виконання збігається з величиной виділеного кванта, тому процес працює до свого завершення. Тепер черга процесів в змозі готовність складається з двох процесів, p2 і p0. Процесор виділяється процесу p2. Він завершується до закінчення відпущеного йому процесорного часу, і чергові кванти відмірюються процесу p0 – що не єдиному закінчив до цього моменту свою роботу. Час очікування для процесу p0 (кількість символів "Г" у відповідному рядку) складає 5 одиниць часу, для процесу p1 – 4 одиниці часу, для процесу p2 – 8 одиниць часу. Таким чином, середній час очікування для цього алгоритму виходить рівним (5 + 4 + 8) /3 = 5,6(6) одиниць часу. Повний час виконання для процесу p0 (кількість непорожніх стовпців у відповідному рядку) складає 18 одиниць часу, для процесу p1 – 8 одиниць, для процесу p2 – 9 одиниць. Середній повний час виконання виявляється рівним (18 + 8 + 9) /3 = 11,6(6) одиниць часу.

Легко побачити, що середній час очікування і середній повний час виконання для зворотного порядку процесів не відрізняються від відповідних часів для алгоритму FCFS і складають 2 і 6 одиниць часу відповідно.

На продуктивність алгоритму RR сильно впливає величина кванта часу. Розглянемо той же самий приклад з порядком процесів p0, p1, p2 для величини кванта часу, рівної 1 (див. таблиця 3.3). Час очікування для процесу p0 складе 5 одиниць часу, для процесу p1 – теж 5 одиниць, для процесу p2 – 2 одиниці. В цьому випадку середній час очікування виходить рівним (5 + 5 + 2) /3 = 4 одиницям часу. Середній повний час виконання складе (18 + 9 + 3) /3 = 10 одиниць часу.

Таблиця 3.3.

Час	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	І	Грам	Грам	І	Грам	І	Грам	І	Грам	І	І	І	І	І	І	І	І	І
p1	Грам	І	Грам	Грам	І	Грам	І	Грам	І									
p2	Грам	Грам	І															

При дуже великих величинах кванта часу, коли кожен процес встигає завершити свій CPU burst до виникнення переривання за часом, алгоритм RR вироджується в алгоритм FCFS. При дуже малих величинах створюється ілюзія того, що кожен з n процесів працює на власному віртуальному процесорі з продуктивністю ~ 1/n від продуктивності реального процесора. Правда, це справедливо лише при теоретичному аналізі за умови зневаги часом перемикання контексту процесів. У реальних умовах при дуже малій величині кванта часу і, відповідно, дуже частому перемиканні контексту накладні витрати на перемикання різко знижують продуктивність системи.

Модуль 2. Управління процесами в операційних системах.

Shortest-Job-First (SJF)

При розгляді алгоритмів FCFS і RR ми бачили, наскільки істотним для них є порядок розташування процесів в черзі процесів, готових до виконання. Якщо короткі завдання розташовані в черзі ближче до її початку, то загальна продуктивність цих алгоритмів значно зростає. Якби ми знали час наступних CPU burst для процесів, що знаходяться в змозі готовність, то могли б вибрати для виконання не процес з початку черги, а процес з мінімальною тривалістю CPU burst. Якщо ж таких процесів два або більше, то для вибору одного з них можна використовувати вже відомий нам алгоритм FCFS. Квантування часу при цьому не застосовується. Описаний алгоритм отримав назву "Найкоротша робота першої" або Shortest Job First (SJF).

SJF-алгоритм короткострокового планування може бути таким, що як витісняє, так і не витісняє. При невитісняючому SJF-плануванні процесор надається вибраному процесу на все необхідне йому час, незалежно від подій, що відбуваються в обчислювальній системі. При витісняючому SJF-плануванні враховується поява нових процесів в черзі готових до виконання (з числа що знов народилися або розблокованих) під час роботи вибраного процесу. Якщо CPU burst нового процесу менше, ніж залишок CPU burst у того, що виконується, то процес, що виконується, витісняється новим.

Розглянемо приклад роботи невитісняючого алгоритму SJF. Хай в змозі готовність знаходяться чотири процеси, p0, p1, p2 і p3, для яких відомі часи їх чергових CPU burst. Ці часи приведені в таблиця 4.1. Як і раніше, вважатимемо, що вся діяльність процесів обмежується використанням тільки одного проміжку CPU burst, що процеси не здійснюють операцій введення-виводу і що часом перемикання контексту можна нехтувати.

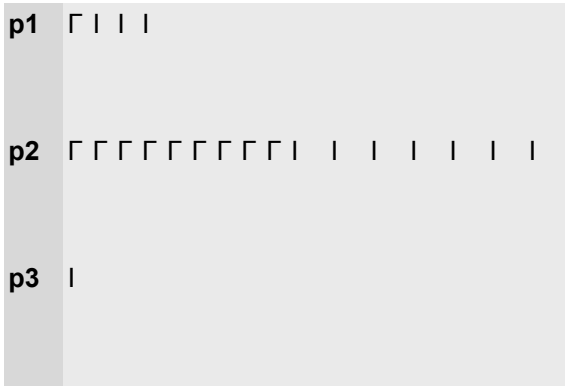
Таблиця 4.1.

Процес	p0	p1	p2	p3
Тривалість чергового CPU burst	5	3	7	1

При використанні невитісняючого алгоритму SJF першим для виконання буде вибраний процес p3, що має найменше значення тривалості чергового CPU burst. Після його завершення для виконання вибирається процес p1, потім p0 і, нарешті, p2. Ця картина відбита в таблиця 4.2.

Таблиця 4.2.

Час	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p0	Г	Г	Г	Г	І	І	І	І	І							



Як ми бачимо, середній час очікування для алгоритму SJF складає (4 + 1 + 9 + 0) /4 = 3,5 одиниць часу. Легко порахувати, що для алгоритму FCFS при порядку процесів p0, p1, p2, p3 ця величина дорівнюватиме (0 + 5 + 8 + 15) /4 = 7 одиницям часу, тобто буде в два рази більше, ніж для алгоритму SJF. Можна показати, що для заданого набору процесів (якщо в черзі не з'являються нові процеси) алгоритм SJF є оптимальним з погляду мінімізації середнього часу очікування серед класу невитісняючих алгоритмів.

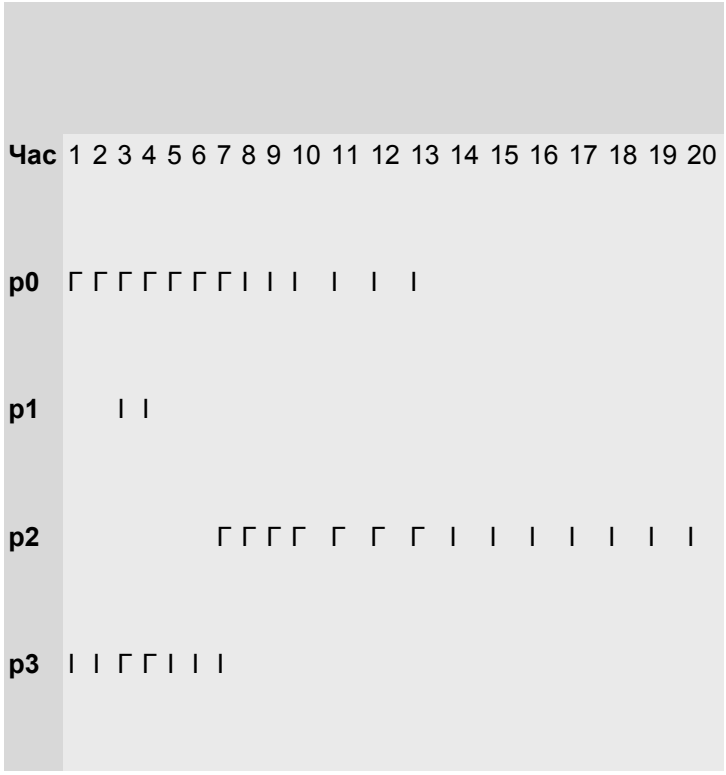
Для розгляду прикладу витісняючого планування SJF ми візьмемо ряд процесів p0, p1, p2 і p3 з різними часом CPU burst і різними моментами їх появи в черзі процесів, готових до виконання (див. табл. 4.3).

Таблиця 4.3.

Процес	Час появи в черзі чергового CPU burst	Тривалість
p0	0	6
p1	2	2
p2	6	7
p3	0	5

У початковий момент часу в змозі готовність знаходяться тільки два процеси, p0 і p3. Менший час чергового CPU burst опиняється біля процесу p3, тому він і вибирається для виконання (див. таблиця 4.4). Після 2 одиниць часу в систему поступає процес p1. Час його CPU burst менше, ніж залишок CPU burst у процесу p3, який витісняється із стану виконання і переводиться в стан готовність. Після закінчення ще 2 одиниць часу процес p1 завершується, і для виконання знов вибирається процес p3. У момент часу t = 6 в черзі процесів, готових до виконання, з'являється процес p2, але оскільки йому для роботи потрібно 7 одиниць часу, а процесу p3 залишилося трудитися всього 1 одиницю часу, то процес p3 залишається в змозі виконання. Після його завершення у момент часу t = 7 в черзі знаходяться процеси p0 і p2, з яких вибирається процес p0. Нарешті, останнім дістане можливість виконуватися процес p2.

Таблиця 4.4.



Основну складність при реалізації алгоритму SJF представляє неможливість точного знання тривалості чергового CPU burst для процесів, що виконуються. У пакетних системах кількість процесорного часу, необхідне завданню для виконання, указує користувач при формуванні завдання. Ми можемо брати цю величину для здійснення довгострокового SJF-планування. Якщо користувач вкаже більше часу, чим йому потрібно, він чекатиме результату довше, ніж міг би, оскільки завдання буде завантажено в систему пізніше. Якщо ж він вкаже меншу кількість часу, завдання може не долічитися до кінця. Таким чином, в пакетних системах рішення задачі оцінки часу використання процесора перекладається на плечі користувача. При короткостроковому плануванні ми можемо робити тільки прогноз тривалості наступного CPU burst, виходячи з передісторії роботи процесу. Хай ((n) – величина n-го CPU burst, T(n+1) – значення, що передбачається, для n+1-го CPU burst (– деяка величина в діапазоні від 0 до 1.

Визначимо рекурентне співвідношення

T(n+1)= (((n)+(1-()T(n)

T(0) покладемо довільною константою. Перший доданок враховує остання поведінка процесу, тоді як другий доданок враховує його передісторію. При (= 0 ми перестаємо стежити за останньою поведінкою процесу, фактично вважаючи

T(n)= T(n+1) =...=T(0)

тобто оцінюючи все CPU burst однаково, виходячи з деякого початкового припущення.

Поклавши (= 1, ми забуваємо про передісторію процесу. В цьому випадку ми вважаємо, що час чергового CPU burst збігатиметься з часом останнього CPU burst:

T(n+1)= ((n)

Зазвичай вибирають (= 1/2 для рівноцінного обліку останньої поведінки і передісторії. Треба відзначити, що такий вибір (зручний і для швидкої організації обчислення оцінки T(n+1). Для підрахунку нової оцінки потрібно узяти стару оцінку, скласти із зміряним часом CPU burst і отриману суму розділити на 2, наприклад, зрушивши її на 1 битий управо. Отримані оцінки

T(n+1) застосовуються як тривалість чергових проміжків часу безперервного використання процесора для короткострокового SJF-планирования.

Гарантоване планування

При інтерактивній роботі N користувачів в обчислювальній системі можна застосувати алгоритм планування, який гарантує, що кожен з користувачів матиме в своєму розпорядженні ~1/N частина процесорного часу. Пронумеруємо всіх користувачів від 1 до N. Для кожного користувача з номером і введемо дві величини: Ti – час знаходження користувача в системі або, іншими словами, тривалість сеансу його спілкування з машиною і (i – сумарний процесорний час вже виділений всім його процесам протягом сеансу. Справедливим для користувача було б отримання Ti/N процесорного часу. Якщо

$\tau_{i<<Ti/N}$

то і-й користувач несправедливо обділений процесорним часом. Якщо ж

$\tau_{i>>Ti/N}$

то система явно благоволить до користувача з номером і. Обчислимо для процесів кожного користувача значення коефіцієнта справедливості

$\tau_{iN/Ti}$

і надаватимемо черговий квант часу готовому процесу з найменшою величиною цього відношення. Запропонований алгоритм називають алгоритмом гарантованого планування. До недоліків цього алгоритму можна віднести неможливість передбачити поведінку користувачів. Якщо деякий користувач відправиться на пару годин пообідати і поспати, не перериваючи сеансу роботи, то після повернення його процеси отримуватимуть невиправдано багато процесорного часу.

Пріоритетне планування

Алгоритмами SJF і гарантованого планування є окремі випадки пріоритетного планування. При пріоритетному плануванні кожному процесу привласнюється певне числове значення – пріоритет, відповідно до якого йому виділяється процесор. Процеси з однаковими пріоритетами плануються в порядку FCFS. Для алгоритму SJF як такий пріоритет виступає оцінка тривалості наступного CPU burst. Чим менше значення цієї оцінки, тим більше високий пріоритет має процес. Для алгоритму гарантованого планування пріоритетом служить обчислений коефіцієнт справедливості. Чим він менший, тим більше у процесу пріоритет.

Алгоритми призначення пріоритетів процесів можуть спиратися як на внутрішні параметри, пов'язані з тим, що відбувається усередині обчислювальної системи, так і на зовнішніх по відношенню до неї. До внутрішніх параметрів відносяться різні кількісні і якісні характеристики процесу такі як: обмеження за часом використання процесора, вимоги до розміру пам'яті, число відкритих файлів і використовуваних пристроїв введення-виводу, відношення середньої тривалості I/O burst до CPU burst і так далі Алгоритми SJF і гарантованого планування використовують внутрішні параметри. Як зовнішні параметри можуть виступати важливість процесу для досягнення яких-небудь цілей, вартість сплаченого процесорного часу, і інші політичні чинники. Високий зовнішній пріоритет може бути привласнений завданню лектора або того, хто заплатив $100 за роботу протягом однієї години.

Планування з використанням пріоритетів може бути таким, що як витісняє, так і не витісняє. При витісняючому плануванні процес з вищим пріоритетом, що з'явився в черзі готових

процесів, витісняє процес, що виконується, з нижчим пріоритетом. У разі невитісняючого планування він просто стає в початок черги готових процесів. Давайте розглянемо приклади використання різних режимів пріоритетного планування.

Хай в чергу процесів, що знаходяться в змозі готовність, поступають ті ж процеси, що і в прикладі для витісняючого алгоритму SJF, тільки їм додатково ще привласнені пріоритети (див. таблиця 4.5). У обчислювальних системах не існує певної угоди, яке значення пріоритету – 1 або 4 вважати за більш пріоритетний. Щоб уникнути плутанини, у всіх наших прикладах ми припускатимемо, що більше значення відповідає меншому пріоритету, тобто найбільш пріоритетним в нашому прикладі є процес p3, а найменш пріоритетним – процес p0.

Таблиця 4.5.

Процес	Час появи в черзі	Тривалість чергового CPU burst	Пріоритет
p0	0	6	4
p1	2	2	3
p2	6	7	2
p3	0	5	1

Як поводитимуться процеси при використанні невитісняючого пріоритетного планування? Першим для виконання у момент часу t = 0 вибирається процес p3, як що володіє найвищим пріоритетом. Після його завершення у момент часу t = 5 в черзі процесів, готових до виконання, опиняться два процеси p0 і p1. Більший пріоритет з них у процесу p1, він і почне виконуватися (див. таблиця 4.6). Потім у момент часу t = 8 для виконання буде вибраний процес p2, і лише потім – процес p0.

Таблиця 4.6.

Час	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	І	І	І	І	І	І
p1			Г	Г	Г	І	І													
p2							Г	І	І	І	І	І	І	І						

p3	I I I I I

Іншим буде надання процесора процесам у разі витісняючого пріоритетного планування (див. таблиця 4.7). Першим, як і у попередньому випадку, почне виконуватися процес p3, а після його закінчення – процес p1. Проте у момент часу t = 6 він буде витиснений процесом p2 і продовжить своє виконання тільки у момент часу t = 13. Останнім, як і раніше, виконуватиметься процес p0.

Таблиця 4.7.

Час	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	I	I	I	I	I	I	
p1			Г	Г	Г	I	Г	Г	Г	Г	Г	Г	Г	I						
p2								I	I	I	I	I	I	I						
p3	I	I	I	I	I															

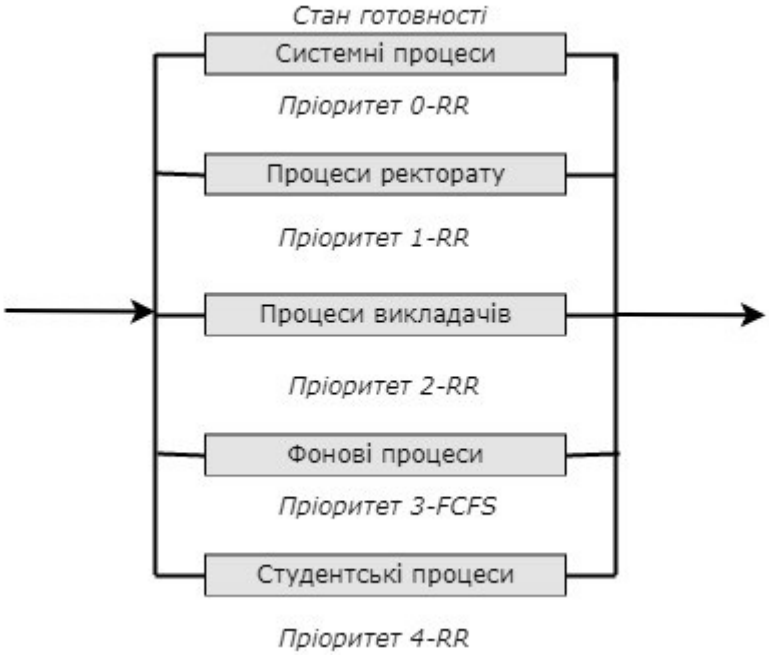
У розглянутому вище прикладі пріоритети процесів з часом не змінювалися. Такі пріоритети прийнято називати статичними. Механізми статичної пріоритетності легко реалізувати, і вони зв'язані з відносно невеликими витратами на вибір найбільш пріоритетного процесу. Проте статичні пріоритети не реагують на зміни ситуації в обчислювальній системі, які можуть зробити бажаною коректування порядку виконання процесів. Гнучкішими є динамічні пріоритети процесів, що змінюють свої значення по ходу виконання процесів. Початкове значення динамічного пріоритету, привласнене процесу, діє протягом лише короткого періоду часу, після чого йому призначається нове, більш відповідне значення. Зміна динамічного пріоритету процесу є єдиною операцією над процесами, яку ми до цих пір не розглянули. Як правило, зміна пріоритету процесів проводиться погоджено із здійсненням яких-небудь інших операцій: при народженні нового процесу, при розблокуванні або блокуванні процесу, після закінчення певного кванта часу або після закінчення процесу. Прикладами алгоритмів з динамічними пріоритетами є алгоритм SJF і алгоритм гарантованого планування. Схеми з динамічною пріоритетністю набагато складніше в реалізації і пов'язані з великими витратами в порівнянні із статичними схемами. Проте їх використання припускає, що ці витрати виправдовуються поліпшенням роботи системи.

Головна проблема пріоритетного планування полягає в тому, що при неналежному виборі механізму призначення і зміни пріоритетів низько пріоритетні процеси можуть не запускатися невизначено довгий час. Зазвичай трапляється одне з двох. Або вони все ж таки чекають своєї черги на виконання (у дев'ять годинників ранку в неділю, коли всі пристойні програмісти лягають спати). Або обчислювальну систему доводиться вимикати, і вони втрачаються (при

зупинці IBM 7094 в Массачусетському технологічному інституті в 1973 році були знайдені процеси, що запущені в 1967 році і жодного разу з тих пір не виконувалися). Вирішення цієї проблеми може бути досягнуте за допомогою збільшення з часом значення пріоритету процесу, що знаходиться в змозі готовність. Хай спочатку процесам привласнюються пріоритети від 128 до 255. Кожного разу після закінчення певного проміжку часу значення пріоритетів готових процесів зменшуються на 1. Процесу, що побував в змозі виконання, привласнюється первинне значення пріоритету. Навіть така груба схема гарантує, що будь-якому процесу в розумні терміни буде надано право на виконання.

Багаторівневі черги (Multilevel Queue)

Для систем, в яких процеси можуть бути легко розсортовані по різних групах, був розроблений інший клас алгоритмів планування. Для кожної групи процесів створюється своя черга процесів, що знаходяться в змозі готовність (див. мал. 4.1). Цим чергам приписуються фіксовані пріоритети. Наприклад, пріоритет черги системних процесів встановлюється вище, ніж пріоритет черг призначених для користувача процесів. А пріоритет черги процесів, запущених студентами, нижче, ніж для черги процесів, запущених викладачами. Це означає, що жоден призначений для користувача процес не буде вибраний для виконання, поки є хоч один готовий системний процес, і жоден студентський процес не отримає в своє розпорядження процесор, якщо є процеси викладачів, готові до виконання. Усередині цих черг для планування можуть застосовуватися самі різні алгоритми. Так, наприклад, для великих рахункових процесів, що не вимагають взаємодії з користувачем (фонових процесів), може використовуватися алгоритм FCFS, а для інтерактивних процесів – алгоритм RR. Подібний підхід, що отримав назву багаторівневих черг, підвищує гнучкість планування: для процесів з різними характеристиками застосовується найбільш відповідний ним алгоритм.

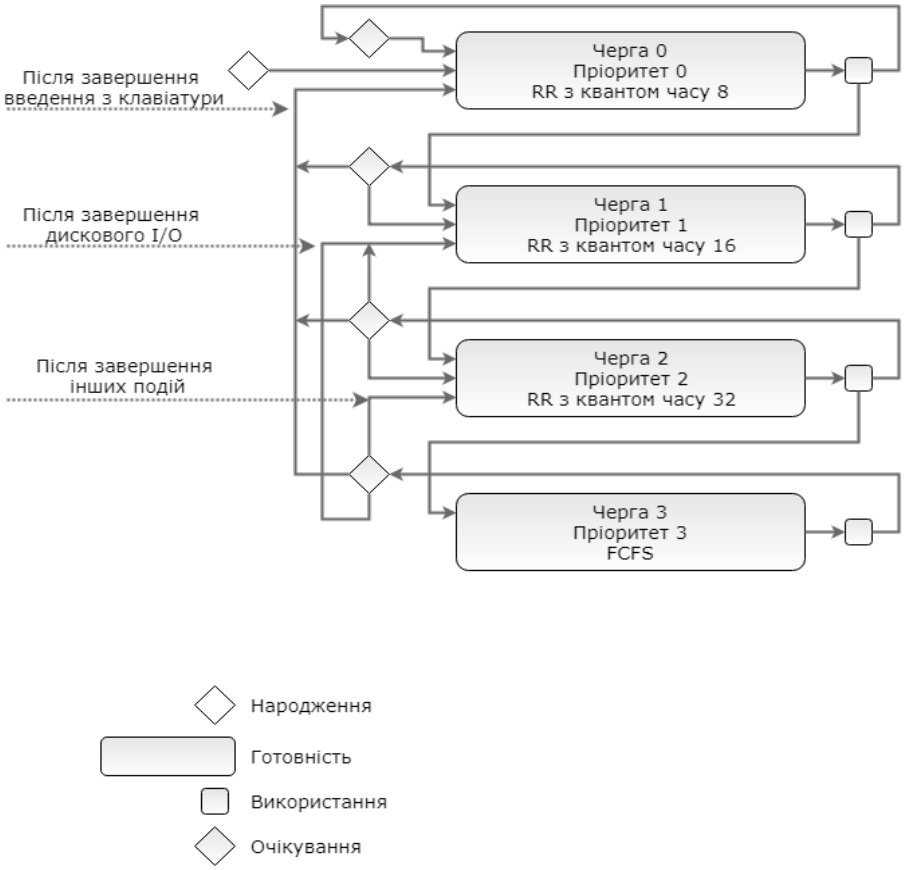


Мал. 4.1. Декілька черг планування

Багаторівневі черги із зворотним зв'язком (Multilevel Feedback Queue)

Подальшим розвитком алгоритму багаторівневих черг є додавання до нього механізму зворотного зв'язку. Тут процес не постійно приписаний до певної черги, а може мігрувати з однієї черги в іншу залежно від своєї поведінки.

Для простоти розглянемо ситуацію, коли процеси в змозі готовність організовані в 4 черги, як на мал. 4.2. Планування процесів між чергами здійснюється на основі витісняючого пріоритетного механізму. Чим вище на малюнку розташовується черга, тим вище її пріоритет. Процеси в черзі 1 не можуть виконуватися, якщо в черзі 0 є хоч би один процес. Процеси в черзі 2 не будуть вибрані для виконання, поки є хоч один процес в чергах 0 і 1. І нарешті, процес в черзі 3 може отримати процесор в своє розпорядження тільки тоді, коли черги 0, 1 і 2 порожні. Якщо при роботі процесу з'являється інший процес в якій-небудь пріоритетнішій черзі, процес, що виконується, витісняється новим. Планування процесів усередині черг 0–2 здійснюється з використанням алгоритму RR, планування процесів в черзі 3 ґрунтується на алгоритмі FCFS.



Мал. 4.2. Схема міграції процесів в багаторівневих чергах планування із зворотним зв'язком. Витіснення процесів більш пріоритетними процесами і завершення процесів на схемі не показане

Процес, що народився, поступає в чергу 0. При виборі на виконання він отримує в своє розпорядження квант часу розміром 8 одиниць. Якщо тривалість його CPU burst менше цього кванта часу, процес залишається в черзі 0. Інакше він переходить в чергу 1. Для процесів з черги 1 квант часу має величину 16. Якщо процес не укладається в цей час, він переходить в чергу 2. Якщо укладається – залишається в черзі 1. У черзі 2 величина кванта часу складає 32 одиниці. Якщо для безперервної роботи процесу і цього мало, процес поступає в чергу 3, для якої квантування часу не застосовується і, за відсутності готових процесів в інших чергах, може виконуватися до закінчення свого CPU burst. Чим більше значення тривалості CPU burst, тим в менш пріоритетну чергу потрапляє процес, але тим на більший процесорний час він може розраховувати. Таким чином, через деякий час всі процеси, що вимагають малого часу роботи процесора, виявляться розміщеними у високо пріоритетних чергах, а всі процеси, що вимагають великого рахунку і з низькими запитами до часу відгуку, – в низько пріоритетних.

Міграція процесів у зворотному напрямі може здійснюватися по різних принципах. Наприклад, після завершення очікування введення з клавіатури процеси з черг 1, 2 і 3 можуть поміщатися в чергу 0, після завершення дискових операцій введення-виводу процеси з черг 2 і 3 можуть поміщатися в чергу 1, а після завершення очікування всіх інших подій – з черги 3 в чергу 2. Переміщення процесів з черг з низькими пріоритетами в черзі з високими пріоритетами дозволяє більш повно враховувати зміну поведінки процесів з часом.

Багаторівневі черги із зворотним зв'язком є найбільш загальним підходом до планування процесів з числа підходів, розглянутих нами. Вони найбільш важкі в реалізації, але в той же час володіють найбільшою гнучкістю. Зрозуміло, що існує багато інших різновидів такого способу планування, окрім варіанту, приведеного вище. Для повного опису їх конкретного втілення необхідно вказати:

- Кількість черг для процесів, що знаходяться в змозі готовність.
- Алгоритм планування, що діє між чергами.
- Алгоритми планування, що діють усередині черг.
- Правила приміщення процесу, що народився, в одну з черг.
- Правила перекладу процесів з однієї черги в іншу.

Змінюючи який-небудь з перерахованих пунктів, ми можемо істотно міняти поведінку обчислювальної системи.

На цьому ми припиняємо розгляд різних алгоритмів планування процесів, бо, як було сказано: "Не можна обійняти неосяжне".

Висновок

Одним з найбільш обмежених ресурсів обчислювальної системи є процесорний час. Для його розподілу між численними процесами в системі доводиться застосовувати процедуру планування процесів. По ступеню тривалості впливу планування на поведінку обчислювальної системи розрізняють короткострокове, середньострокове і довгострокове планування процесів. Конкретні алгоритми планування процесів залежать від поставлених цілей, класу вирішуваних завдань і спираються на статичні і динамічні параметри процесів і комп'ютерних систем. Розрізняють витісняючий і невитісняючий режими планування. При невитісняючому плануванні процесом, що виконується, поступається процесор іншому процесу тільки за власним бажанням, при витісняючому плануванні процес, що виконується, може бути витиснений із стану виконання окрім своєї волі.

Простим алгоритмом планування є невитісняючий алгоритм FCFS, який, проте, може істотно затримувати короткі процеси, що не вчасно перейшли в стан готовність. У системах розділення часу широкого поширення набула витісняюча версія цього алгоритму – RR.

Серед всіх невитісняючих алгоритмів оптимальним з погляду середнього часу очікування процесів є алгоритм SJF. Існує і витісняючий варіант цього алгоритму. У інтерактивних системах часто використовується алгоритм гарантованого планування, що забезпечує користувачам рівні частини процесорного часу.

Алгоритм SJF і алгоритм гарантованого планування є окремими випадками планування з використанням пріоритетів. У більш загальних методах пріоритетного планування застосовуються багаторівневі черги процесів, готових до виконання, і багаторівневі черги із зворотним зв'язком. Будучи найбільш складними в реалізації, ці способи планування забезпечують гнучка поведінка обчислювальних систем і їх адаптивність до вирішення завдань різних класів.

Взаємодіючі процеси

Для досягнення поставленої мети різні процеси (можливо, що навіть належать різним користувачам) можуть виконуватися псевдопараллельно на одній обчислювальній системі або паралельно на різних обчислювальних системах, взаємодіючи між собою.

Для чого процесам потрібно займатися спільною діяльністю? Які існують причини для їх кооперації?

- Підвищення швидкості роботи. Поки один процес чекає настання деякої події (наприклад, закінчення операції введення-виводу), інші можуть займатися корисною роботою, направленою на рішення загальної задачі. У багатопроцесорних обчислювальних системах програма розбивається на окремі шматочки, кожен з яких виконуватиметься на своєму процесорі.
- Сумісне використання даних. Різні процеси можуть, наприклад, працювати з однією і тією ж динамічною базою даних або з файлом, що розділяється, спільно змінюючи їх вміст.
- Модульна конструкція якої-небудь системи. Типовим прикладом може служити мікроядерний спосіб побудови операційної системи, коли різними її частинами є окремі процеси, що взаємодіють шляхом передачі повідомлень через мікроядро.
- Нарешті, це може бути необхідно просто для зручності роботи користувача, охочого, наприклад, редагувати і відладжувати програму одночасно. У цій ситуації процеси редактора і відладчика повинні уміти взаємодіяти один з одним.

Процеси не можуть взаємодіяти, не спілкуючись, тобто не обмінюючись інформацією. "Спілкування" процесів зазвичай приводить до зміни їх поведінки залежно від отриманої інформації. Якщо діяльність процесів залишається незмінною при будь-якій прийнятій ними інформації, то це означає, що вони насправді "спілкування" не потребують. Процеси, які впливають на поведінку один одного шляхом обміну інформацією, прийнято називати кооперативними або взаємодіючими процесами, на відміну від незалежних процесів, що не надають один на одного ніякої дії.

Різні процеси в обчислювальній системі спочатку є відособленою суттю. Робота одного процесу не повинна приводити до порушення роботи іншого процесу. Для цього, зокрема, розділені їх адресні простори і системні ресурси, і для забезпечення коректної взаємодії процесів потрібні спеціальні засоби і дії операційної системи. Не можна просто помістити значення, обчислене в одному процесі, в область пам'яті, відповідну змінною в іншому процесі, не зробивши яких-небудь додаткових зусиль. Давайте розглянемо основні аспекти організації спільної роботи процесів.

Категорії засобів обміну інформацією

Процеси можуть взаємодіяти один з одним, тільки обмінюючись інформацією. За об'ємом інформації, яка передається, і ступеню можливої дії на поведінку іншого процесу всі засоби такого обміну можна розділити на три категорії.

- Сигнальні. Передається мінімальна кількість інформації – один біт, "та чи ні". Використовуються, як правило, для сповіщення процесу про настання якої-небудь події. Ступінь дії на поведінку процесу, що отримав інформацію, мінімальна. Все залежить від того, чи знає він, що означає отриманий сигнал, чи треба на нього реагувати і яким

чином. Неправильна реакція на сигнал або його ігнорування можуть привести до трагічних наслідків. Пригадаємо професора Плейшнера з кінофільму "Сімнадцять митей весни". Сигнал тривоги – квітковий горщик на підвіконні – був йому переданий, але професор проігнорував його. І до чого це привело?

- Канальні. "Спілкування" процесів відбувається через лінії зв'язки, надані операційною системою, і нагадує спілкування людей по телефону, за допомогою записок, листів або оголошень. Об'єм інформації, яка передається, в одиницю часу обмежений пропускною спроможністю ліній зв'язку. Із збільшенням кількості інформації зростає і можливість впливу на поведінку іншого процесу.
- Пам'ять, що розділяється. Два або більш за процеси можуть спільно використовувати деяку область адресного простору. Створенням пам'яті, що розділяється, займається операційна система (якщо, звичайно, її про це попросять). "Спілкування" процесів нагадує сумісне мешкання студентів в одній кімнаті гуртожитку. Можливість обміну інформацією максимальна, як, втім, і вплив на поведінку іншого процесу, але вимагає підвищеної обережності (якщо ви переклали на інше місце речі вашого сусіда по кімнаті, а частину з них ще і викинули). Використання пам'яті, що розділяється, для передачу/отримання інформації здійснюється за допомогою засобів звичайних мов програмування, тоді як сигнальним і канальним засобам комунікації для цього необхідні спеціальні системні виклики. Пам'ять, що розділяється, є найбільш швидким способом взаємодії процесів в одній обчислювальній системі.

Нитки виконання

Розглянуті вище аспекти логічної реалізації відносяться до засобів зв'язку, орієнтованих на організацію взаємодії різних процесів. Проте зусилля, направлені на прискорення вирішення завдань в рамках класичних операційних систем, привели до появи абсолютно інших механізмів, до зміни самого поняття "процес".

Свого часу впровадження ідеї мультипрограмування дозволило підвищити пропускну спроможність комп'ютерних систем, тобто зменшити середній час очікування результатів роботи процесів. Але будь-який окремо узятий процес в мультипрограмній системі ніколи не може бути виконаний швидше, ніж при роботі в однопрограмному режимі на тому ж обчислювальному комплексі. Проте, якщо алгоритм вирішення задачі володіє певним внутрішнім паралелізмом, ми могли б прискорити його роботу, організувавши взаємодію декількох процесів. Розглянемо наступний приклад. Хай у нас є наступна програма на псевдомові програмування:

Ввести масив a

Ввести масив b

Ввести масив

a = a + b

з = a + з

Вивести масив

При виконанні такої програми в рамках одного процесу цей процес чотири рази блокуватиметься, чекаючи закінчення операцій введення-виводу. Але наш алгоритм володіє внутрішнім паралелізмом. Обчислення суми масивів a + b можна було б виконувати паралельно з очікуванням закінчення операції введення масиву c.

Ввести масив a

Очікування закінчення операції введення

Ввести масив b

Очікування закінчення операції введення

Ввести масив

Очікування закінчення операції введення a = a + b

з = a + з

Вивести масив

Очікування закінчення операції виводу

Таке поєднання операцій за часом можна було б реалізувати, використовуючи два взаємодіючі процеси. Для простоти вважатимемо, що засобом комунікації між ними служить пам'ять, що розділяється. Тоді наші процеси можуть виглядати таким чином.

Процес 1	Процес 2
Ввести масив a	Очікування введення
Очікування закінчення операції введення	масивів a і b
Ввести масив b	
Очікування закінчення операції введення	
Ввести масив	
Очікування закінчення операції введення	a = a + b
з = a + з	
Вивести масив	
Очікування закінчення операції виводу	

Здавалося б, ми запропонували конкретний чин прискорення рішення задачі. Проте насправді справа йде не так просто. Другий процес має бути створений, обидва процеси повинні повідомити операційну систему, що їм необхідна пам'ять, яку вони могли б розділити з іншим процесом, і, нарешті, не можна забувати про перемикання контексту. Тому реальна поведінка процесів виглядатиме приблизно так.

Процес 1 Процес 2

Створити процес 2

Перемикання контексту

Виділення загальної

пам'яті

Очікування введення

a і b

Перемикання контексту

Виділення загальній пам'яті

Ввести масив a

Очікування закінчення

операції введення

Ввести масив b

Очікування закінчення

операції введення

Ввести масив

Очікування закінчення

операції введення

Перемикання контексту

a = a + b

Перемикання контексту

з = a + з

Вивести масив

Очікування закінчення

операції виводу

Очевидно, що ми можемо не тільки не виграти в часі при рішенні задачі, але навіть і програти, оскільки тимчасові втрати на створення процесу, виділення загальній пам'яті і перемикання контексту можуть перевищити виграш, отриманий за рахунок поєднання операцій.

Для того, щоб реалізувати нашу ідею, введемо нову абстракцію усередині поняття "процес" – нитка виконання або просто нитка (у англомовній літературі використовується термін thread). Нитки процесу розділяють його програмний код, глобальні змінні і системні ресурси, але кожна нитка має власний програмний лічильник, свій вміст регістрів і свій стек. Тепер процес представляється як сукупність взаємодіючих ниток і виділених йому ресурсів. Процес, що містить всього одну нитку виконання, ідентичний процесу в тому сенсі, який ми вживали раніше. Для таких процесів ми надалі використовуватимемо термін "традиційний процес". Іноді нитки називають полегшеними процесами або міні-процесами, оскільки у багатьох відношеннях вони подібні до традиційних процесів. Нитки, як і процеси, можуть породжувати нитки-нащадки, правда, тільки усередині свого процесу, і переходити з одного стану в інше. Стани ниток аналогічні станам традиційних процесів. Із стану народження процес приходить таким, що містить всього одну нитку виконання. Інші нитки процесу будуть нащадками цієї нитки-прародички. Ми можемо вважати, що процес знаходиться в змозі готовність, якщо хоч би одна з його ниток знаходиться в змозі готовність і жодна з ниток не знаходиться в змозі виконання. Ми можемо вважати, що процес знаходиться в змозі виконання, якщо одна з його ниток знаходиться в змозі виконання. Процес знаходитиметься в змозі очікування, якщо всі його нитки знаходяться в змозі очікування. Нарешті, процес знаходиться в змозі закінчив виконання, якщо всі його нитки знаходяться в змозі закінчила виконання. Поки одна нитка процесу заблокована, інша нитка того ж процесу може виконуватися. Нитки розділяють процесор так само, як це робили традиційні процеси, відповідно до розглянутих алгоритмів планування.

Оскільки нитки одного процесу розділяють істотно більше ресурсів, чим різні процеси, то операції створення нової нитки і перемикання контексту між нитками одного процесу займають значно менше часу, чим аналогічні операції для процесів в цілому. Запропонована нами схема поєднання роботи в термінах ниток одного процесу отримує право на існування.

Нитка 1 Нитка 2

Створити нитку 2

Перемикання контексту ниток

Очікування введення a і b

Перемикання контексту ниток

Ввести масив a

Очікування закінчення

операції введення

Ввести масив b

Очікування закінчення

операції введення

Ввести масив

Очікування закінчення

операції введення

Перемикання контексту ниток

a = a + b

Перемикання контексту ниток

з = а + з

Вивести масив

Очікування закінчення

операції виводу

Розрізняють операційні системи, що підтримують нитки на рівні ядра і на рівні бібліотек. Все сказане вище справедливо для операційних систем, що підтримують нитки на рівні ядра. У них планування використання процесора відбувається в термінах ниток, а управління пам'яттю і іншими системними ресурсами залишається в термінах процесів. У операційних системах, що підтримують нитки на рівні бібліотек користувачів, і планування процесора, і управління системними ресурсами здійснюються в термінах процесів. Розподіл використання процесора по нитках в рамках виділеного процесу тимчасового інтервалу здійснюється засобами бібліотеки. У подібних системах блокування однієї нитки приводить до блокування всього процесу, бо ядро операційної системи не має уявлення про існування ниток. По суті справи, в таких обчислювальних системах просто імітується наявність ниток виконання.

Далі в цій частині книги для простоти викладу ми використовуватимемо термін "процес", хоча все сказане відноситиметься і до ниток виконання.

Висновок

Для досягнення поставленої мети різні процеси можуть виконуватися псевдопаралельно на одній обчислювальній системі або паралельно на різних обчислювальних системах, взаємодіючи між собою. Причинами для спільної діяльності процесів зазвичай є: необхідність прискорення рішення задачі, сумісне використання оновлюваних даних, зручність роботи або модульний принцип побудови програмних комплексів. Процеси, які впливають на поведінку один одного шляхом обміну інформацією, називають кооперативними або взаємодіючими процесами, на відміну від незалежних процесів, що не надають один на одного ніякої дії і нічого що не знають про взаємне існування в обчислювальній системі.

Для забезпечення коректного обміну інформацією операційна система повинна надати процесам спеціальні засоби зв'язку. За об'ємом інформації, яка передається, і ступеню можливої дії на поведінку процесу, що отримав інформацію, їх можна розділити на три категорії: сигнальні, канальні і пам'ять, що розділяється. Через канальні засоби комунікації інформація може передаватися у вигляді потоку даних або у вигляді повідомлень і накопичуватися в буфері певного розміру. Для ініціалізації "спілкування" процесів і його припинення можуть потрібно спеціальні дії з боку операційної системи. Процеси, зв'язуючись один з одним, можуть використовувати непряму, пряму симетричну і пряму асиметричну схеми адресації. Існують одно- і двонаправлені засоби передачі інформації. Засоби комунікації забезпечують надійний зв'язок, якщо при спілкуванні процесів не відбувається втрати і пошкодження інформації, не з'являється зайвій інформації, не порушується порядок даних.

Зусилля, направлені на прискорення вирішення завдань в рамках класичних операційних систем, привели до появи нової абстракції усередині поняття "процес" – нитки виконання або просто нитки. Нитки процесу розділяють його програмний код, глобальні змінні і системні

ресурси, але кожна нитка має власний програмний лічильник, свій вміст регістрів і свій стек. Тепер процес представляється як сукупність взаємодіючих ниток і виділених йому ресурсів. Нитки можуть породжувати нові нитки усередині свого процесу, вони мають стани, аналогічні станам процесу, і можуть переводитися операційною системою з одного стану в інше. У системах, що підтримують нитки на рівні ядра, планування використання процесора здійснюється в термінах ниток виконання, а управління рештою системних ресурсів – в термінах процесів. Накладні витрати на створення нової нитки і на перемикання контексту між нитками одного процесу істотно менше, ніж на ті ж самі дії для процесів, що дозволяє на однопроцесорній обчислювальній системі прискорювати вирішення завдань за допомогою організації роботи декількох взаємодіючих ниток.

Алгоритми синхронізації

Interleaving, race condition і взаємовиключання

Давайте тимчасово відвернемося від операційних систем, процесів і ниток виконання і поговоримо про деякі "активності". Під активностями ми розумітимемо послідовне виконання ряду дій, направлених на досягнення певної мети. Активності можуть мати місце в програмному і технічному забезпеченні, в звичайній діяльності людей і тварин. Ми розбиватимемо активності на деякі неділимі, або атомарні, операції. Наприклад, активність "приготування бутерброда" можна розбити на наступні атомарні операції:

1. Відрізувати скибочку хліба.
2. Відрізувати скибочку ковбаси.
3. Намазати скибочку хліба маслом.
4. Покласти скибочку ковбаси на підготовлену скибочку хліба.

Неподільні операції можуть мати внутрішні невидимі дії (узяти батон хліба в ліву руку, узяти ніж в праву руку, провести відрізання). Ми ж називаємо їх неділимими тому, що вважаємо за виконуваних за раз, без переривання діяльності.

Хай є дві активність

P: a b з

Q: d e f

де a, b, з, d, e, f – атомарні операції. При послідовному виконанні активностей ми отримуємо таку послідовність атомарних дій:

PQ: a b з d e f

Що відбудеться при виконання цих активностей псевдопаралельно, в режимі розділення часу? Активності можуть розшаруватися на неподільні операції з різним чергуванням, тобто може відбутися те, що англійською мовою прийнято називати словом interleaving. Можливі варіанти чергування:

a b з d e f

a b d з e f

a b d e з f

a b d e f з

a d b з e f

.....

d e f a b з

Атомарні операції активностей можуть чергуватися всілякими різними способами із збереженням порядку розташування усередині активностей. Оскільки псевдопаралельне виконання два активностей приводить до чергування їх неподільних операцій, результат псевдопаралельного виконання може відрізнятися від результату послідовного виконання. Розглянемо приклад. Хай у нас є дві активність P і Q, що складаються з двох атомарних операцій кожна:

P: x=2 Q: x=3

y=x-1 y=x+1

Що ми отримаємо в результаті їх псевдопаралельного виконання, якщо змінні x і y є для активностей загальними? Очевидно, що можливі чотири різних набору значень для пари (x, y): (3, 4), (2, 1), (2, 3) і (3, 2). Ми говоритимемо, що набір активностей (наприклад, програм) детермінований, якщо всякий раз при псевдопаралельного виконання для одного і того ж набору вхідних даних він дає однакові вихідні дані. Інакше він недетермінований. Вище приведений приклад недетермінованого набору програм. Зрозуміло, що детермінований набір активностей можна небоязливо виконувати в режимі розділення часу. Для недетермінованого набору такого виконання небажане.

Чи можна до отримання результатів визначити, чи є набір активностей детермінованим чи ні? Для цього існують достатні умови Бернстайна. Викладемо їх стосовно програм із змінними, що розділяються.

Введемо набори вхідних і вихідних змінних програми. Для кожної атомарної операції набори вхідних і вихідних змінних – це набори змінних, які атомарна операція прочитує і записує. Набор вхідних змінних програми R(P) (R від слова read) суть об'єднання наборів вхідних змінних для всіх її неподільних дій. Аналогічно, набір вихідних змінних програми W(P) (W від слова write) суть об'єднання наборів вихідних змінних для всіх її неподільних дій. Наприклад, для програми

P: x=u+v

y=x*w

отримуємо R(P)= {u, v, x, w}, W(P)= {x, y}. Відмітимо, що змінна x присутня як в R(P), так і в W(P).

Тепер сформулюємо умови Бернстайна.

Якщо для двох даних активностей P і Q:

- перетин W(P) і W(Q) порожньо
- перетин W(P) з R(Q) порожньо
- перетин R(P) і W(Q) порожньо

тоді виконання P і Q детерміноване.

Якщо ці умови не дотримані, можливо, паралельне виконання P і Q детерміноване, а може бути, і ні.

Випадок два активності природним чином узагальнюються на їх більшу кількість.

Умови Бернстайна інформативні, але дуже жорсткі. По суті справи, вони вимагають практично невзаємодіючих процесів. А нам хотілося б, щоб детермінований набір утворювали активності, що спільно використовують інформацію і що обмінюються нею. Для цього нам необхідно обмежити число можливих чергувань атомарних операцій, виключивши деякі чергування за допомогою механізмів синхронізації виконання програм, забезпечивши тим самим впорядкований доступ програм до деяких даних.

Про недетермінований набір програм (і активності взагалі) говорять, що він має race condition (стан гонки, стан змагання). У приведеному вище прикладі процеси змагаються за обчислення значень змінних x і y.

Завдання впорядкованого доступу до даних (усунення race condition), що розділяються, у тому випадку, коли нам не важлива його черговість, можна вирішити, якщо забезпечити кожному процесу ексклюзивне право доступу до цих даних. Кожен процес, що звертається до ресурсів, що розділяються, унеможливлює для всіх інших процесів одночасного спілкування з цими ресурсами, якщо це може привести до недетермінованої поведінки набору процесів. Такий прийом називається таким, що взаємовиключає (mutual exclusion). Якщо черговість доступу до ресурсів, що розділяються, важлива для отримання правильних результатів, то одними взаємовиключаннями вже не обійтися, потрібна взаимосинхронизация поведінки програм.

Критична секція

Важливим поняттям при вивченні способів синхронізації процесів є поняття критичної секції (critical section) програми. Критична секція – це частина програми, виконання якої може привести до виникнення race condition для певного набору програм. Щоб виключити ефект гонок по відношенню до деякого ресурсу, необхідно організувати роботу так, щоб в кожен момент часу тільки один процес міг знаходитися в своїй критичній секції, пов'язаній з цим ресурсом. Іншими словами, необхідно забезпечити реалізацію того, що взаємовиключає для критичних секцій програм. Реалізація того, що взаємовиключає для критичних секцій програм з практичної точки зору означає, що по відношенню до інших процесів, що беруть участь у взаємодії, критична секція починає виконуватися як атомарна операція. Давайте розглянемо наступний приклад, в якому псевдопаралельні взаємодіючі процеси представлені діями різних студентів (таблиця 5.1):

Тут критична ділянка для кожного процесу – від операції "Виявляє, що хліба немає" до операції "Повертається в кімнату" включно. В результаті відсутності того, що взаємовиключає ми з ситуації "Немає хліба" потрапляємо в ситуацію хліб "Дуже Багато". Якби ця критична ділянка виконувалася як атомарна операція – "Дістає два батони хліба", то проблема утворення надлишків була б знята.

Таблиця 5.1.

Час	Студент 1	Студент 2	Студент 3
17-05	Приходить в кімнату		
17-07	Виявляє, що хліба немає		
17-09	Йде в магазин		
17-11		Приходить в кімнату	

17-13		Виявляє, що хліба немає	
17-15		Йде в магазин	
17-17			Приходить в кімнату
17-19			Виявляє, що хліба немає
17-21			Йде в магазин
17-23	Приходить в магазин		
17-25	Купує 2 батони на всіх		
17-27	Йде з магазина		
17-29		Приходить в магазин	
17-31		Купує 2 батони на всіх	
17-33		Йде з магазина	
17-35			Приходить в магазин
17-37			Купує 2 батони на всіх
17-39			Йде з магазина
17-41	Повертається в кімнату		
17-43			
17-45			
17-47		Повертається в кімнату	
17-49			
17-51			
17-53			Повертається в кімнату

Зробити процес добування хліба атомарною операцією можна було б таким чином: перед початком цього процесу закрити двері зсередини на засув і йти здобувати хліб через вікно, а після закінчення процесу повернутися в кімнату через вікно і відсунути засув. Тоді поки один студент здобуває хліб, всі останні знаходяться в стані очікування під дверима (таблиця 5.2).

Таблиця 5.2.

Час	**Студент 1**	**Студент 2**	**Студент 3**
17-05	Приходить в кімнату		
17-07	Дістає два батони хліба		
17-43		Приходить в кімнату	
17-47			Приходить в кімнату

Отже, для вирішення завдання необхідно, щоб у тому випадку, коли процес знаходиться в своїй критичній ділянці, інші процеси не могли увійти до своїх критичних ділянок. Ми бачимо, що

критична ділянка повинна супроводжуватися прологом (entry section) – "закрити двері зсередини на засув" – і епілогом (exit section) – "відсунути засув", які не мають відношення до активності одиночного процесу. Під час виконання прологу процес винен, зокрема, отримати дозвіл на вхід в критичну ділянку, а під час виконання епілогу – повідомити інші процеси, що він покинув критичну секцію.

У загальному випадку структура процесу, що бере участь у взаємодії, може бути представлена таким чином:

```
while (some condition) {
  entry section
    critical section
  exit section
    remainder section
}
```

Тут під remainder section розуміються всі атомарні операції, що не входять в критичну секцію.

Частина цієї лекції, що залишилася, присвячена різним способам програмної організації прологу і епілогу критичної ділянки у разі, коли черговість доступу до критичної ділянки не має значення.

Програмні алгоритми організації взаємодії процесів

Вимоги, що пред'являються до алгоритмів

Організація того, що взаємовиключає для критичних ділянок, звичайно, дозволить уникнути виникнення race condition, але не є достатньою для правильної і ефективної паралельної роботи кооперативних процесів. Сформулюємо п'ять умов, які повинні виконуватися для хорошого програмного алгоритму організації взаємодії процесів, що мають критичні ділянки, якщо вони можуть проходити їх в довільному порядку.

1. Завдання має бути вирішена чисто програмним способом на звичайній машині, що не має спеціальних команд того, що взаємовиключає. При цьому передбачається, що основні інструкції мови програмування (такі примітивні інструкції, як load, store, test) є атомарними операціями.
2. Не повинно існувати ніяких припущень про відносні швидкості процесів, що виконуються, або число процесорів, на яких вони виконуються.
3. Якщо процес Pi виконується в своїй критичній ділянці, то не існує ніяких інших процесів, які виконуються у відповідних критичних секціях. Цю умову отримала назва умови того, що взаємовиключає (mutual exclusion).
4. Процеси, які знаходяться поза своїми критичними ділянками і не збираються входити в них, не можуть перешкоджати іншим процесам входити в їх власні критичні ділянки. Якщо немає процесів в критичних секціях і є процеси, охочі увійти до них, то тільки ті процеси, які не виконуються в remainder section, повинні ухвалювати рішення про те, який процес увійде до своєї критичної секції. Таке рішення не повинне ухвалюватися нескінченно довго. Цю умову отримала назва умови прогресу (progress).
5. Не повинно виникати необмежено довгого очікування для входу одного з процесів в свою критичну ділянку. Від того моменту, коли процес запросив дозвіл на вхід в критичну секцію, і до того моменту, коли він цей дозвіл отримав, інші процеси можуть пройти через свої критичні ділянки лише обмежене число разів. Цю умову отримала назва умови обмеженого очікування (bound waiting).

Треба відмітити, що опис відповідного алгоритму в нашому випадку означає опис способу організації прологу і епілогу для критичної секції.

Заборона переривань

Найбільш простим рішенням поставленої задачі є наступна організація прологу і епілогу:

```
while (some condition) {
  заборонити всі переривання
    critical section
  вирішити всі переривання
    remainder section
}
```

Оскільки вихід процесу із стану виконання без його завершення здійснюється по перериванню, усередині критичної секції ніхто не може втрутитися в його роботу. Проте таке рішення може мати наслідки, що далеко йдуть, оскільки дозволяє процесу користувача вирішувати і забороняти переривання у всій обчислювальній системі. Допустимо, що користувач випадково або по злому наміру заборонив переривання в системі і зациклив або завершив свій процес. Без перезавантаження системи в такій ситуації не обійтися.

Проте заборона і дозвіл переривань часто застосовуються як пролог і епілог до критичних секцій усередині самої операційної системи, наприклад при оновленні вмісту PCB.

Змінна-замок

Як наступна спроба рішення задачі для призначених для користувача процесів розглянемо іншу пропозицію. Візьмемо деяку змінну, доступну всім процесам, з початковим значенням рівним 0. Процес може увійти до критичної секції тільки тоді, коли значення цієї змінної-замку дорівнює 0, одночасно змінюючи її значення на 1 – закриваючи замок. При виході з критичної секції процес скидає її значення в 0 – замок відкривається (як у випадку з покупкою хліба студентами в розділі "Критична секція").

```
shared int lock = 0;

/* shared означає, що */

/* змінна є такою, що розділяється */


while (some condition) {

  while(lock); lock = 1;

     critical section

  lock = 0;

     remainder section

}
```

На жаль, при уважному розгляді ми бачимо, що таке рішення не задовольняє умові того, що взаємовиключає, оскільки дія while(lock); lock = 1; не є атомарним. Допустимо, процес P0 протестував значення змінної lock і ухвалив рішення рухатися далі. У цей момент, ще до привласнення змінній lock значення 1, планувальник передав управління процесу P1. Він теж вивчає вміст змінної lock і теж ухвалює рішення увійти до критичної ділянки. Ми отримуємо два процеси, що одночасно виконують свої критичні секції.

Строге чергування

Спробуємо вирішити задачу спочатку для двох процесів. Черговий підхід також використовуватиме загальну для них обох змінну з початковим значенням 0. Тільки тепер вона гратиме не роль замку для критичної ділянки, а явно указувати, хто може наступним увійти до нього. Для i-го процесу це виглядає так:

```
shared int turn = 0;


while (some condition) {
```

```
  while(turn != i);
    critical section
  turn = 1-i;
    remainder section
}
```

Очевидно, що те, що взаємовиключає гарантується, процеси входять в критичну секцію строго по черзі: P0, P1, P0, P1, P0 ... Але наш алгоритм не задовольняє умові прогресу. Наприклад, якщо значення turn дорівнює 1, і процес P0 готовий увійти до критичної ділянки, він не може зробити цього, навіть якщо процес P1 знаходиться в remainder section.

Прапори готовності

Недолік попереднього алгоритму полягає в тому, що процеси нічого не знають про стан один одного у нинішній момент часу. Давайте спробуємо виправити цю ситуацію. Хай два наші процесу мають масив прапорів готовності входу процесів, що розділяється, в критичну ділянку

```
shared int ready[2]= {0, 0};
```

Коли i-й процес готовий увійти до критичної секції, він привласнює елементу масиву ready[i] значення рівне 1. Після виходу з критичної секції він, природно, скидає це значення в 0. Процес не входить в критичну секцію, якщо інший процес вже готовий до входу в критичну секцію або знаходиться в ній.

```
while (some condition) {
  ready[i]= 1;
  while(ready[1-i]);
    critical section
  ready[i]= 0;
    remainder section
}
```

Отриманий алгоритм забезпечує те, що взаємовиключає, дозволяє процесу, готовому до входу в критичну ділянку, увійти до нього відразу після завершення епілогу в іншому процесі, але все одно порушує умову прогресу. Хай процеси практично одночасно підійшли до виконання прологу. Після виконання привласнення ready[0]=1 планувальник передав процесор від процесу 0 процесу 1, який також виконав привласнення ready[1]=1. Після цього обидва процеси нескінченні довго чекають один одного на вході в критичну секцію. Виникає ситуація, яку прийнято називати тупиковою (deadlock). (Докладніше про тупикові ситуації розповідається в лекції 7.)

Алгоритм Петерсона

Перше вирішення проблеми, що задовольняє всім вимогам і що використовує ідеї раніше розглянутих алгоритмів, було запропоноване данським математиком Деккером (Dekker). У 1981 році Петерсон (Peterson) запропонував витонченіше рішення. Хай обидва процеси мають доступ до масиву прапорів готовності і до змінної черговості.

```
shared int ready[2]= {0, 0};
shared int turn;
while (some condition) {
  ready[i]= 1;
  turn =1-i;
  while(ready[1-i]&& turn == 1-i);
    critical section
  ready[i]= 0;
    remainder section
}
```

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

Давайте доведемо, що все п'ять наших вимог до алгоритму дійсно задовольняються.

Задоволення вимог 1 і 2 очевидно.

Доведемо виконання умови того, що взаємовиключає методом від осоружного. Хай обидва процеси одночасно опинилися усередині своїх критичних секцій. Відмітимо, що процес Pi може увійти до критичної секції, тільки якщо ready[1-i]== 0 або turn == i. Відмітимо також, що якщо обидва процеси виконують свої критичні секції одночасно, то значення прапорів готовності для обох процесів збігаються і рівні 1. Чи могли обидва процеси увійти до критичних секцій із стану, коли вони обидва одночасно знаходилися в процесі виконання циклу while? Ні, оскільки в цьому випадку змінна turn повинна була б одночасно мати значення 0 і 1 (коли обидва процеси виконують цикл, значення змінних змінитися не можуть). Хай процес P0 першим увійшов до критичної ділянки, тоді процес P1 повинен був виконати перед входженням в цикл while принаймні один передуючий оператор (turn = 0;). Проте після цього він не може вийти з циклу до закінчення критичної ділянки процесу P0, оскільки при вході в цикл ready[0]== 1 і turn == 0, і ці значення не можуть змінитися до тих пір, поки процес P0 не покине своя критична ділянка. Ми прийшли до суперечності. Отже, має місце те, що взаємовиключає.

Доведемо виконання умови прогресу. Візьмемо, без обмеження спільності, процес P0. Відмітимо, що він не може увійти до своєї критичної секції тільки при сумісному виконанні умов ready[1]== 1 і turn == 1. Якщо процес P1 не готовий до виконання критичної ділянки, то ready[1]== 0, і процес P0 може здійснити вхід. Якщо процес P1 готовий до виконання критичної ділянки, то ready[1]== 1 і змінна turn має значення 0 або 1, дозволяючи процесу P0 або процесу P1 почати виконання критичної секції. Якщо процес P1 завершив виконання критичної ділянки, то він скине свій прапор готовності ready[1]== 0, дозволяючи процесу P0 приступити до виконання критичної роботи. Таким чином, умова прогресу виконується.

Звідси ж витікає виконання умови обмеженого очікування. Оскільки в процесі очікування дозволу на вхід процес P0 не змінює значення змінних, він зможе почати виконання своєї критичної ділянки не більше ніж одного проходу по критичній секції процесу P1.

Алгоритм булочної (Bakery algorithm)

Алгоритм Петерсона дає нам рішення задачі коректної організації взаємодії двох процесів. Давайте розглянемо тепер відповідний алгоритм для n взаємодіючих процесів, який отримав назву алгоритм булочної, хоча стосовно наших умов його слід було б швидше назвати алгоритм реєстратури в поліклініці. Основна його ідея виглядає так. Кожен клієнт (він же процес), що знов прибуває, отримує талончик на обслуговування з номером. Клієнт з найменшим номером на талончику обслуговується наступним. На жаль, із-за неатомарності операції обчислення наступного номера алгоритм булочної не гарантує, що у всіх процесів будуть талончики з різними номерами. У разі рівності номерів на талончиках у двох або більш за клієнтів першим обслуговується клієнт з меншим значенням імені (імена можна порівнювати в лексикографічному порядку). Структури даних, що розділяються, для алгоритму – це два масиви

shared enum {false, true} choosing[n];

shared int number[n];

Спочатку елементи цих масивів ініціюються значеннями false і 0 відповідно. Введемо наступні позначення

(a,b) < (c,d), якщо a < з

або якщо a == з і b < d

max(a0, a1 ...., an) – це число до таке, що

до >= ai для всіх i = 0 ...,n

Структура процесу Pi для алгоритму булочної приведена нижче

```
while (some condition) {
  choosing[i]= true;
  number[i]= max(number[0] ...,
              number[n-1]) + 1;
  choosing[i]= false;
  for(j = 0; j < n; j++){
    while(choosing[j]);
    while(number[j]!= 0 && (number[j],j) <
      (number[i],i));
  }
    critical section
  number[i]= 0;
```

```
    remainder section
}
```

Доказ того, що цей алгоритм задовольняє умовам 1 – 5, виконаєте самостійно як вправу.

Наявність апаратної підтримки тих, що взаємовиключають дозволяє спростити алгоритми і підвищити їх ефективність точно так, як і це відбувається і в інших областях програмування. Ми вже зверталися до загальноприйнятого hardware для вирішення завдання реалізації тих, що взаємовиключають, коли говорили про використання механізму заборони/дозволу переривань.

Багато обчислювальних систем окрім цього мають спеціальні команди процесора, які дозволяють перевірити і змінити значення машинного слова або поміняти місцями значення двох машинних слів в пам'яті, виконуючи ці дії як атомарні операції. Давайте обговоримо, як концепції таких команд можуть використовуватися для реалізації тих, що взаємовиключають.

Команда Test-and-Set (перевірити і привласнити 1)

Про виконання команди Test-and-Set, що здійснює перевірку значення логічної змінної з одночасною установкою її значення в 1, можна думати як про виконання функції

```
int Test_and_Set (int *target){
  int tmp = *target;
  *target = 1;
  return tmp;
}
```

З використанням цієї атомарної команди ми можемо модифікувати наш алгоритм для змінної-замку, так щоб він забезпечував ті, що взаємовиключають

```
shared int lock = 0;


while (some condition) {
  while(Test_and_Set(&lock));
    critical section
  lock = 0;
    remainder section
}
```

На жаль, навіть у такому вигляді отриманий алгоритм не задовольняє умові обмеженого очікування для алгоритмів. Подумайте, як його слід змінити для дотримання всіх умов.

Команда Swap (обміняти значення)

Виконання команди Swap, що обмінює два значення, що знаходяться в пам'яті, можна проілюструвати наступною функцією:

```
void Swap (int *a, int *b){
```

```
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Застосовуючи атомарну команду Swap, ми можемо реалізувати попередній алгоритм, ввівши додаткову логічну змінну key, локальну для кожного процесу:

```
shared int lock = 0;
int key;

while (some condition) {
    key = 1;
    do Swap(&lock,&key);
    while (key);
        critical section
    lock = 0;
        remainder section
}
```

Висновок

Послідовне виконання деяких дій, направлених на досягнення певної мети, називається активністю. Активності складаються з атомарних операцій, що виконуються нерозривно, як одиничне ціле. При виконанні декілька активностей в псевдопаралельному режимі атомарні операції різних активностей можуть перемішуватися між собою з дотриманням порядку проходження всередині активностей. Це явище отримала назва interleaving (чергування). Якщо результати виконання декількох активностей не залежать від варіанту чергування, то такий набір активностей називається детермінованим. Інакше він носить назву недетерміновану. Існує достатня умова Бернстайна для визначення детермінованої набору активностей, але воно накладає дуже жорсткі обмеження на набір, вимагаючи практично не взаємодіючих активностей. Про недетермінований набір активностей говорять, що він має race condition (умова гонки, змагання). Усунення race condition можливо при обмеженні допустимих варіантів чергувань атомарних операцій за допомогою синхронізації поведінки активностей. Ділянки активностей, виконання яких може привести до race condition, називають критичними ділянками. Необхідною умовою для усунення race condition є організація того, що взаємовиключає на критичних ділянках: всередині відповідних критичних ділянок не може одночасно знаходитися більш за одну активність.

Для ефективних програмних алгоритмів усунення race condition окрім умови того, що взаємовиключає потрібне виконання наступних умов: алгоритми не використовують спеціальних команд процесора для організації тих, що взаємовиключають, алгоритми нічого не знають про швидкості виконання процесів, алгоритми задовольняють умовам прогресу і

обмеженого очікування. Всі ці умови виконуються в алгоритмі Петерсона для двох процесів і алгоритмі булочної – для декількох процесів.

Застосування спеціальних команд процесора, що виконують ряд дій як атомарну операцію, – Test-and-Set, Swap – дозволяє істотно спростити алгоритми синхронізації процесів.

Механізми синхронізації

Розглянуті в кінці попередньої лекції алгоритми хоча і є коректними, але достатньо громіздкі і не володіють елегантністю. Більш того, процедура очікування входу в критичну ділянку припускає достатньо тривале обертання процесу в порожньому циклі, тобто марну витрату дорогоцінного часу процесора. Існують і інші серйозні недоліки у алгоритмів, побудованих засобами звичайних мов програмування. Допустимо, що в обчислювальній системі знаходяться два взаємодіючі процеси: один з них – H – з високим пріоритетом, інший – L – з низьким пріоритетом. Хай планувальник влаштований так, що процес з високим пріоритетом витісняє низькопріоритетний процес всякий раз, коли він готовий до виконання, і займає процесор на весь час свого CPU burst (якщо не з'явиться процес з ще більшим пріоритетом). Тоді у випадку, якщо процес L знаходиться в своїй критичній секції, а процес H, отримавши процесор, підійшов до входу в критичну область, ми отримуємо тупикову ситуацію. Процес H не може увійти до критичної області, знаходячись в циклі, а процес L не отримує управління, щоб покинути критичну ділянку.

Для того, щоб не допустити виникнення подібних проблем, були розроблені різні механізми синхронізації більш високого рівня. Опису ряду з них – семафорів, моніторів і повідомлень – і присвячена дана лекція.

Семафори

Одним з перших механізмів, запропонованих для синхронізації поведінки процесів, стали семафори, концепцію яких описав Дейкстра (Dijkstra) в 1965 році.

Концепція семафорів

Семафор є цілу змінну, що набуває ненегативних значень, доступ будь-якого процесу до якої, за винятком моменту її ініціалізації, може здійснюватися тільки через дві атомарні операції: P (від данського слова proberen – перевіряти) і V (від verhogen – збільшувати). Класичне визначення цих операцій виглядає таким чином:

P(S): поки S ? 0 процес блокується;

S = S – 1;

V(S): S = S + 1;

Цей запис означає наступне: при виконанні операції P над семафором S спочатку перевіряється його значення. Якщо воно більше 0, то з S віднімається 1. Якщо воно менше або рівне 0, то процес блокується до тих пір, поки S не стане більше 0, після чого з S віднімається 1. При виконанні операції V над семафором S до його значення просто додається 1. У момент створення семафор може ініціалізувати будь-яким ненегативним значенням.

Подібні змінні-семафори можуть з успіхом застосовуватися для вирішення різних завдань організації взаємодії процесів. У ряді мов програмування вони були безпосередньо введені в синтаксис мови (наприклад, в ALGOL-68), в інших випадках реалізуються за допомогою спеціальних системних викликів. Відповідна ціла змінна розташовується усередині адресного простору ядра операційної системи. Операційна система забезпечує атомарність операцій P і V, використовуючи, наприклад, метод заборони переривань на час виконання відповідних системних викликів. Якщо при виконанні операції P заблокованими виявилися декілька процесів, то порядок їх розблокування може бути довільним, наприклад, FIFO.

Вирішення проблеми producer-consumer за допомогою семафорів

Одному з типових завдань, що вимагають організації взаємодії процесів, є завдання producer-consumer (виробник-споживач). Хай два процеси обмінюються інформацією через буфер обмеженого розміру. Виробник закладає інформацію в буфер, а споживач витягує її звідти. На цьому рівні діяльність споживача і виробника можна описати таким чином.

```
Producer:  while(1){
             produce_item;
             put_item;
           }
Consumer:  while(1){
             get_item;
             consume_item;
           }
```

Якщо буфер заповнений, то виробник повинен чекати, поки в нім з'явиться місце, щоб покласти туди нову порцію інформації. Якщо буфер порожній, то споживач повинен чекати нового повідомлення. Як можна реалізувати ці умови за допомогою семафорів? Візьмемо три семафори: empty, full і mutex. Семафор full використовуватимемо для гарантії того, що споживач чекатиме, поки в буфері з'явиться інформація. Семафор empty використовуватимемо для організації очікування виробника при заповненому буфері, а семафор mutex – для організації того, що взаємовиключає на критичних ділянках, якими є дії put_item і get_item (операції «покласти інформацію» і «узяти інформацію» не можуть перетинатися, оскільки в цьому випадку виникне небезпека спотворення інформації). Тоді рішення задачі на С-подобном мові виглядає так:

```
  Semaphore mutex = 1;
  Semaphore empty = N; /* де N – ємкість буфера*/
  Semaphore full = 0;
Producer:
  while(1){
    produce_item;
    P(empty);
    P(mutex);
    put_item;
    V(mutex);
    V(full);
  }
Consumer:
  while(1){
```

```
    P(full);
    P(mutex);
    get_item;
    V(mutex);
    V(empty);
    consume_item;
  }
```

Легко переконатися, що це дійсно коректне рішення поставленої задачі. Попутно відмітимо, що семафори використовувалися тут для досягнення двох цілей: організації того, що взаємовиключає на критичній ділянці і взаимосинхронизации швидкості роботи процесів.

Монітори

Хоча рішення задачі producer-consumer за допомогою семафорів виглядає достатньо витончено, програмування з їх використанням вимагає підвищеної обережності і уваги, ніж частково нагадує програмування на мові Асемблера. Допустимо, що в розглянутому прикладі ми випадково поміняли місцями операції P, спочатку виконавши операцію для семафора mutex, а вже потім для семафорів full і empty. Допустимо тепер, що споживач, увійшовши до своєї критичної ділянки (mutex скинутий), виявляє, що буфер порожній. Він блокується і починає чекати появи повідомлень. Але виробник не може увійти до критичної ділянки для передачі інформації, оскільки той заблокований споживачем. Отримуємо тупикову ситуацію.

У складних програмах провести аналіз правильності використання семафорів з олівцем в руках стає дуже непросто. В той же час звичайні способи відладки програм часто не дають результату, оскільки виникнення помилок залежить від interleaving атомарних операцій, і помилки можуть бути трудновоспроизводимы. Для того, щоб полегшити роботу програмістів, в 1974 році Хором (Hoare) був запропонований механізм ще більш високого рівня, ніж семафори, що отримав назву моніторів. Ми з вами розглянемо конструкцію, декілька що відрізняється від оригінальної.

Монітори є типом даних, який може бути з успіхом упроваджений в об'єктно-орієнтовані мови програмування. Монітор володіє власними змінними, що визначають його стан. Значення цих змінних ззовні можуть бути змінені тільки за допомогою виклику функцій-методів, що належать монітору. У свою чергу, ці функції-методи можуть використовувати в роботі тільки дані, що знаходяться усередині монітора, і свої параметри. На абстрактному рівні можна описати структуру монітора таким чином:

```
monitor monitor_name {
  опис внутрішніх змінних ;

  void m1(...){...
  }
  void m2(...){...
  }
```

```
    ...
    void mn(...){...
    }

    {
      блок ініціалізації
      внутрішніх змінних;
    }
  }
```

Тут функції m1..., mn є функціями-методами монітора, а блок ініціалізації внутрішніх змінних містить операції, які виконуються один і лише один раз: при створенні монітора або при найпершому виклику якої-небудь функції-методу до її виконання.

Важливою особливістю моніторів є те, що у будь-який момент часу тільки один процес може бути активний, тобто знаходитися в змозі готовність або виконання, усередині даного монітора. Оскільки моніторами є особливі конструкції мови програмування, компілятор може відрізнити виклик функції, що належить монітору, від викликів інших функцій і обробити його спеціальним чином, додавши до нього пролог і епілог, що реалізовує те, що взаємовиключає. Оскільки обов'язок конструювання механізму тих, що взаємовиключають покладений на компілятор, а не на програміста, робота програміста при використанні моніторів істотно спрощується, а вірогідність виникнення помилок стає менше.

Проте одних тільки взаємовиключають недостатньо для того, щоб в повному об'ємі реалізувати вирішення завдань, що виникають при взаємодії процесів. Нам потрібні ще і засоби організації черговості процесів, подібно до семафорів full і empty в попередньому прикладі. Для цього в моніторах було введено поняття умовних змінних (condition variables) (у деяких російських виданнях їх ще називають змінними стани), над якими можна здійснювати дві операції wait і signal, частково схожі на операції P і V над семафорами.

Якщо функція монітора не може виконуватися далі, поки не наступить деяка подія, вона виконує операцію wait над якою-небудь умовною змінною. При цьому процес, що виконав операцію wait, блокується, стає неактивним, і інший процес отримує можливість увійти до монітора.

Коли очікувана подія відбувається, інший процес усередині функції-методу здійснює операцію signal над тією ж самою умовною змінною. Це приводить до пробудження раніше заблокованого процесу, і він стає активним. Якщо декілька процесів чекали операції signal для цієї змінної, то активним стає тільки один з них. Що можна зробити для того, щоб у нас не опинилося двох процесів, що розбудив і пробужденного, одночасно активних усередині монітора? Хор запропонував, щоб пробужденный процес пригнічував виконання процесу, що розбудив, поки він сам не покине монітор. Декілька пізніше Хансен (Hansen) запропонував інший механізм: процес, що розбудив, покидає монітор негайно після виконання операції signal. Ми дотримуватимемося підходу Хансена.

Необхідно відзначити, що умовні змінні, на відміну від семафорів Дейкстри, не уміють запам'ятовувати передісторію. Це означає, що операція signal завжди повинна виконуватися після операції wait. Якщо операція signal виконується над умовною змінною, з якою не пов'язано

жодного заблокованого процесу, то інформація про подію, що відбулася, буде загублена. Отже, виконання операції wait завжди приводитиме до блокування процесу.

Давайте застосуємо концепцію моніторів до рішення задачі виробник-споживач.

```
monitor ProducerConsumer {
  condition full, empty;
  int count;
  void put() {
    if(count == N) full.wait;
    put_item;
    count += 1;
    if(count == 1) empty.signal;
  }
  void get() {
    if (count == 0) empty.wait;
    get_item();
    count -= 1;
    if(count == N-1) full.signal;
  }
  {
    count = 0;
  }
}
```

```
Producer:
  while(1){
    produce_item;
    ProducerConsumer.put();
  }
Consumer:
  while(1){
    ProducerConsumer.get();
```

```
    consume_item;
  }
```

Легко переконатися, що приведений приклад дійсно вирішує поставлену задачу.

Реалізація моніторів вимагає розробки спеціальних мов програмування і компіляторів для них. Монітори зустрічаються в таких мовах, як паралельний Евклід, паралельний Паськаль, Java і так далі Емуляція моніторів за допомогою системних викликів для звичайних широко використовуваних мов програмування не так проста, як емуляція семафорів. Тому можна користуватися ще одним механізмом з тими, що прихованими взаємовиключають, механізмом, про який ми вже згадували, – передачею повідомлень.

Повідомлення

Для прямої і непрямої адресації досить два примітиви, щоб описати передачу повідомлень по лінії зв'язку – send і receive. У разі прямої адресації ми позначатимемо їх так:

send(P, message) – послати повідомлення message процесу P;

receive(Q, message) – отримати повідомлення message від процесу Q.

У разі непрямої адресації ми позначатимемо їх так:

send(A, message) – послати повідомлення message в поштову скриньку A;

receive(A, message) – отримати повідомлення message з поштової скриньки A.

Примітиви send і receive вже мають прихований від наших очей механізм того, що взаємовиключає. Більш того, в більшості систем вони вже мають і прихований механізм блокування при читанні з порожнього буфера і при записі в повністю заповнений буфер. Реалізація рішення задачі producer-consumer для таких примітивів стає непристойно тривіальною. Треба відзначити, що, не дивлячись на простоту використання, передача повідомлень в межах одного комп'ютера відбувається істотно повільніше, ніж робота з семафорами і моніторами.

Еквівалентність семафорів, моніторів і повідомлень

Ми розглянули три високорівневі механізми, процесів, що використовуються для організації взаємодії. Можна показати, що в рамках однієї обчислювальної системи, коли процеси мають можливість використовувати пам'ять, що розділяється, всі вони еквівалентні. Це означає, що будь-які два із запропонованих механізмів можуть бути реалізовані на базі третього механізму, що залишився.

Реалізація моніторів і передачі повідомлень за допомогою семафорів

Розглянемо спочатку, як реалізувати монітори за допомогою семафорів. Для цього нам потрібно уміти реалізовувати ті, що взаємовиключають при вході в монітор і умовні змінні. Візьмемо семафор mutex з початковим значенням 1 для реалізації того, що взаємовиключає при вході в монітор і по одному семафору сі для кожної умовної змінної. Крім того, для кожної умовної змінної заведемо лічильник fi для індикації наявності чекаючих процесів. Коли процес входить в монітор, компілятор генеруватиме виклик функції monitor_enter, яка виконує операцію P над семафором mutex для даного монітора. При нормальному виході з монітора (тобто при

виході без виклику операції signal для умовної змінної) компілятор генеруватиме виклик функції monitor_exit, яка виконує операцію V над цим семафором.

Для виконання операції wait над умовною змінною компілятор генеруватиме виклик функції wait, яка виконує операцію V для семафора mutex, дозволяючи іншим процесам входити в монітор, і виконує операцію P над відповідним семафором ci, блокуючи процес, що викликав. Для виконання операції signal над умовною змінною компілятор генеруватиме виклик функції signal_exit, яка виконує операцію V над асоційованим семафором ci (якщо є процеси, чекаючі відповідної події), і вихід з монітора, минувши функцію monitor_exit.

```
Semaphore mutex = 1;


void monitor_enter(){
   P(mutex);
}


void monitor_exit(){
   V(mutex);
}


Semaphore ci = 0;
int fi = 0;


void wait(i){
   fi=fi + 1;
   V(mutex);
   P(ci );
   fi=fi - 1;
}


void signal_exit(i){
   if (fi) V(ci );
   else V(mutex);
}
```

Відмітимо, що при виконанні функції signal_exit, якщо хто-небудь чекав цієї події, процес покидає монітор без збільшення значення семафора mutex, не вирішуючи тим самим всім

процесам, окрім розбудженого, увійти до монітора. Це збільшення зробить розбуджений процес, коли покине монітор звичайним способом або коли виконає нову операцію wait над якою-небудь умовною змінною.

Розглянемо тепер, як реалізувати передачу повідомлень, використовуючи семафори. Для простоти опишемо реалізацію тільки одній черги повідомлень. Виділимо в пам'яті, що розділяється, достатньо велику область під зберігання повідомлень, там же записуватимемо, скільки порожніх і заповнених осередків знаходиться в буфері, зберігати посилання на списки процесів, чекаючого читання і пам'яті. Взаємовиключання при роботі з пам'яттю, що розділяється, забезпечуватимемо семафором mutex. Також заведемо по одному семафору сі на взаємодіючий процес, для того, щоб забезпечувати блокування процесу при спробі читання з порожнього буфера або при спробі запису в переповнений буфер. Подивимося, як такий механізм працюватиме. Почнемо з процесу, охочого отримати повідомлення.

Процес-одержувач з номером і перш за все виконує операцію P(mutex), отримуючи в монопольне володіння пам'ять, що розділяється. Після чого він перевіряє, чи є в буфері повідомлення. Якщо немає, то він заносить себе в список процесів, які чекають повідомлення, виконує V(mutex) і P(ci). Якщо повідомлення в буфері є, то він читає його, змінює лічильники буфера і перевіряє, чи є процеси в списку процесів, спраглих записи. Якщо таких процесів немає, то виконується V(mutex), і процес-одержувач виходить з критичної області. Якщо такий процес є (з номером j), то він віддаляється з цього списку, виконується V для його семафора cj, і ми виходимо з критичного району. Процес, що прокинувся, починає виконуватися в критичному районі, оскільки mutex у нас має значення 0 і ніхто більш не може потрапити в критичний район. При виході з критичного району саме розбуджений процес проведе виклик V(mutex).

Як будується робота процесу-відправника з номером i? Процес, що посилає повідомлення, теж чекає, поки він не зможе мати монополії на використання пам'яті, що розділяється, виконавши операцію P(mutex). Далі він перевіряє, чи є місце в буфері, і якщо так, то поміщає повідомлення в буфер, змінює лічильники і дивиться, чи є процеси, чекаючі повідомлення. Якщо немає, виконує V(mutex) і виходить з критичної області, якщо є, «будить» один з них (з номером j), викликаючи V(cj), з одночасним видаленням цього процесу із списку процесів, чекаючих повідомлень, і виходить з критичного регіону без виклику V(mutex), надаючи тим самим можливість розбудженому процесу прочитати повідомлення. Якщо місця в буфері немає, то процес-відправник заносить себе в чергу процесів, які чекають можливості запису, і викликає V(mutex) і P(ci).

Реалізація семафорів і передачі повідомлень за допомогою моніторів

Нам досить показати, що за допомогою моніторів можна реалізувати семафори, оскільки отримувати з семафорів повідомлення ми вже уміємо.

Найпростіший спосіб такої реалізації виглядає таким чином. Заведемо усередині монітора змінну-лічильник, пов'язаний з емульованим семафором список процесів, що блокуються, і по одній умовній змінній на кожен процес. При виконанні операції P над семафором зухвалий процес перевіряє значення лічильника. Якщо воно більше нуля, зменшує його на 1 і виходить з монітора. Якщо воно дорівнює 0, процес додає себе в чергу процесів, чекаючих події, і виконує операцію wait над своєю умовною змінною. При виконанні операції V над семафором процес збільшує значення лічильника, перевіряє, чи є процеси, чекаючі цієї події, і якщо є, видаляє один з них із списку і виконує операцію signal для умовної змінної, відповідної процесу.

Реалізація семафорів і моніторів за допомогою черг повідомлень

Покажемо, нарешті, як реалізувати семафори за допомогою черг повідомлень. Для цього скористаємося хитрішою конструкцією, ввівши новий синхронізуючий процес. Цей процес має лічильник і чергу для процесів, які чекають включення семафора. Для того, щоб виконати операції P і V, процеси посилають синхронізуючому процесу повідомлення, в яких указують свої потреби, після чого чекають отримання підтвердження від синхронізуючого процесу.

Після отримання повідомлення синхронізуючий процес перевіряє значення лічильника, щоб з'ясувати, чи можна зробити необхідну операцію. Операція V завжди може бути виконана, тоді як операція P може зажадати блокування процесу. Якщо операція може бути здійснена, то вона виконується, і синхронізуючий процес посилає підтверджуюче повідомлення. Якщо процес має бути блокований, то його ідентифікатор заноситься в чергу блокованих процесів, і підтвердження не посилається. Пізніше, коли який-небудь з інших процесів виконає операцію V, один з блокованих процесів віддаляється з черги очікування і отримує відповідне підтвердження.

Оскільки ми показали раніше, як з семафорів побудувати монітори, ми довели еквівалентність моніторів, семафорів і повідомлень.

Висновок

Для організації синхронізації процесів можуть застосовуватися спеціальні механізми високого рівня, блокуючі процеси, які чекають входу в критичну секцію або настання своєї черги для використання сумісного ресурсу. До таких механізмів відносяться, наприклад, семафори, монітори і повідомлення. Всі ці конструкції є еквівалентними, використовуючи будь-яку з них, можна реалізувати дві що залишилися.

Тупики

Введення

У попередніх лекціях ми розглядали способи синхронізації процесів, які дозволяють процесам успішно кооперуватися. Проте в деяких випадках можуть виникнути непередбачені утруднення. Припустимо, що декілька процесів конкурують за володіння кінцевим числом ресурсів. Якщо запрошуваний процесом ресурс недоступний, ОС переводить даний процес в стан очікування. У разі коли необхідний ресурс утримується іншим процесом, який чекає, перший процес не зможе змінити свій стан. Така ситуація називається тупиком (deadlock). Говорять, що в мультипрограмній системі процес знаходиться в стані тупику, якщо вона чекає події, яка ніколи не відбудеться. Системна тупикова ситуація, або «зависання системи», є слідством того, що один або більш за процеси знаходяться в стані тупику. Іноді подібні ситуації називають взаємоблокуваннями. У загальному випадку проблема тупику ефективного рішення не має.

Розглянемо приклад. Припустимо, що два процеси здійснюють вивід із стрічки на принтер. Один з них встиг монополізувати стрічку і претендує на принтер, а інший навпаки. Після цього обидва процеси виявляються заблокованими в очікуванні другого ресурсу (див. мал. 8.1).

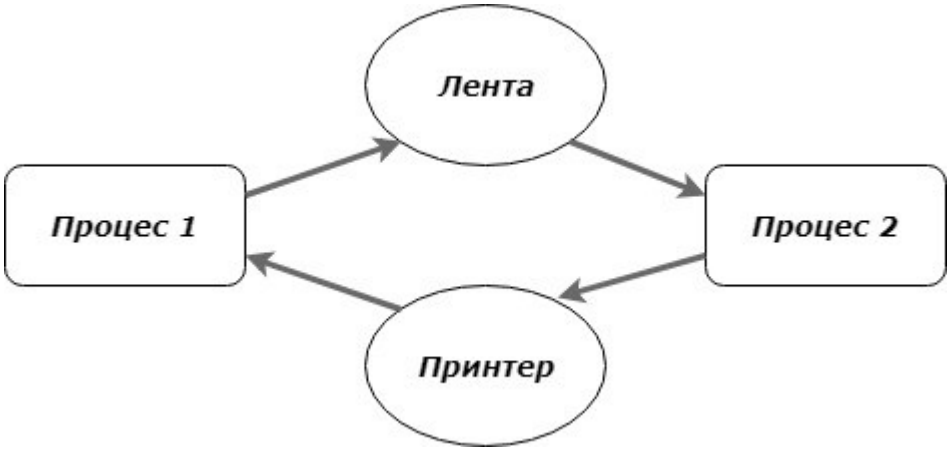


Рис. 8.1. Приклад тупикової ситуації

Визначення. Множина процесів знаходиться в тупиковій ситуації, якщо кожен процес з множини чекає події, яка може викликати тільки інший процес даної множини. Оскільки всі процеси чогось чекають, то жоден з них не зможе ініціювати подію, яка розбудила б іншого члена множини і, отже, всі процеси спатимуть разом.

Вище приведений приклад взаємоблокування, що виникає при роботі з так званими виділеними пристроями. Безвихідь, проте, може мати місце і в інших ситуаціях. Наприклад, в системах управління базами даних запису можуть бути локалізовані процесами, щоб уникнути стану гонок (див. лекцію 5 «Алгоритмів синхронізації»). В цьому випадку може вийти так, що один з процесів заблокував записи, необхідні іншому процесу, і навпаки. Таким чином, безвихідь може мати місце як на апаратних, так і на програмних ресурсах.

Безвихідь також може бути викликані помилками програмування. Наприклад, процес може марно чекати відкриття семафора, тому що в некоректно написаному застосуванні цю операцію забули передбачити. Іншою причиною нескінченного очікування може бути дискримінаційна політика по відношенню до деяких процесів. Проте найчастіше подія, якої чекає процес в

тупиковій ситуації, – звільнення ресурсу, тому надалі будуть розглянуті методи боротьби з безвихіддю ресурсного типу.

Ресурсами можуть бути як пристрої, так і дані. Деякі ресурси допускають розділення між процесами, тобто є ресурсами, що розділяються . Наприклад, пам'ять, процесор, диски колективно використовуються процесами. Інші не допускають розділення, тобто є виділеними, наприклад стрічкопротяжний пристрій. До взаємоблокування може привести використання як виділених, так і таких, що розділяються ресурсів. Наприклад, читання з диска, що розділяється, може одночасно здійснюватися декількома процесами, тоді як запис припускає винятковий доступ до даних на диску. Можна вважати, що частина диска, куди відбувається запис, виділена конкретному процесу. Тому надалі ми виходитимемо з припущення, що безвихідь пов'язана з виділеними ресурсами, тобто безвихідь виникає, коли процесу надається ексклюзивний доступ до пристроїв, файлів і інших ресурсів.

Традиційна послідовність подій при роботі з ресурсом складається із запиту, використання і звільнення ресурсу. Тип запиту залежить від природи ресурсу і від ОС. Запит може бути явним, наприклад спеціальний виклик request, або неявним – open для відкриття файлу. Зазвичай, якщо ресурс зайнятий і запит відхилений, що запрошує процес переходить в стан очікування.

Далі в даній лекції розглядатимуться питання виявлення, запобігання, обходу безвиході і відновлення після безвиході. Як правило, боротьба з безвихіддю – дуже дорогий захід. Проте для ряду систем, наприклад для систем реального часу, іншого виходу немає.

Умови виникнення тупиків

Умови виникнення безвиході були сформульовані Коффманом, Елфіком і Шошані в 1970 р.

1. Умова того, що взаємовиключає (Mutual exclusion). Одночасно використовувати ресурс може тільки один процес.
2. Умова очікування ресурсів (Hold and wait). Процеси утримують ресурси, вже виділені їм, і можуть запрошувати інші ресурси.
3. Умова неперерозподіленості (No preemtion). Ресурс, виділений раніше, не може бути примусово забраний у процесу. Звільнені вони можуть бути тільки процесом, який їх утримує.
4. Умова кругового очікування (Circular wait). Існує кільцевий ланцюг процесів, в якому кожен процес чекає доступу до ресурсу, що утримується іншим процесом ланцюга.

Для утворення безвиході необхідним і достатнім є виконання всіх чотирьох умов.

Зазвичай безвихідь моделюється циклом в графові, що складається з вузлів двох видів: прямокутників – процесів і еліпсів – ресурсів, на зразок того, що зображений на мал. 7.1. Стрілки, направлені від ресурсу до процесу, показують, що ресурс виділений даному процесу. Стрілки, направлені від процесу до ресурсу, означають, що процес запрошує даний ресурс.

Основні напрями боротьби з тупиками

Проблема безвиході ініціювала багато цікавих досліджень в області інформатики. Очевидно, що умова циклічного очікування відрізняється від останніх. Перші три умови формують правила, що існують в системі, тоді як четверта умова описує ситуацію, яка може скластися при певній несприятливій послідовності подій. Тому методи запобігання взаємоблокувань орієнтовані головним чином на порушення перших трьох умов шляхом введення ряду обмежень на поведінку процесів і способи розподілу ресурсів. Методи

виявлення і усунення менш консервативні і зводяться до пошуку і розриву циклу очікування ресурсів.

Отже, основні напрями боротьби з безвихіддю:

- Ігнорування проблеми в цілому
- Запобігання безвиході
- Виявлення безвиході
- Відновлення після безвиході

Ігнорування проблеми тупиків

Простий підхід – не помічати проблему безвиході. Для того, щоб ухвалити таке рішення, необхідно оцінити вірогідність виникнення взаємоблокування і порівняти її з вірогідністю збитку від інших відмов апаратного і програмного забезпечення. Проектувальники зазвичай не бажають жертвувати продуктивністю системи або зручністю користувачів для впровадження складних і дорогих засобів боротьби з безвихіддю.

Будь-яка ОС, що має в ядрі ряд масивів фіксованої розмірності, потенційно страждає від безвиході, навіть якщо вони не виявлені. Таблиця відкритих файлів, таблиця процесів, фактично кожна таблиця є обмеженими ресурсами. Заповнення всіх записів таблиці процесів може привести до того, що черговий запит на створення процесу може бути відхилений. При несприятливому збігу обставин декілька процесів можуть видати такий запит одночасно і попасти в безвихідь. Чи слід відмовлятися від виклику CreateProcess, щоб вирішити цю проблему?

Підхід більшості популярних ОС (Unix, Windows і ін.) полягає в тому, щоб ігнорувати дану проблему в припущенні, що маловірогідна випадкова безвихідь переважно, чим безглузді правила, що примушують користувачів обмежувати число процесів, відкритих файлів і тому подібне Стикаючись з небажаним вибором між строгістю і зручністю, важко знайти рішення, яке влаштовувало б всіх.

Способи запобігання тупикам

Мета запобігання безвиході – забезпечити умови, що унеможливлюють виникнення тупикових ситуацій. Більшість методів пов'язана із запобіганням одній з умов виникнення взаємоблокування.

Система, надаючи ресурс в розпорядження процесу, повинна ухвалити рішення, безпечно це чи ні. Виникає питання: чи є такий алгоритм, який допомагає завжди уникати безвиході і робити правильний вибір. Відповідь – так, ми можемо уникати безвиході, але тільки якщо певна інформація відома заздалегідь.

Способи запобігання тупикам шляхом ретельного розподілу ресурсів. Алгоритм банкіра

Можна уникнути взаємоблокування, якщо розподіляти ресурси, дотримуючись певних правил. Серед такого роду алгоритмів найбільш відомий алгоритм банкіра, запропонований Дейкстрой, який базується на так званих безпечних або надійних станах (safe state). Безпечний стан – це такий стан, для якого є принаймні одна послідовність подій, яка не приведе до взаємоблокування. Модель алгоритму заснована на діях банкіра, який, маючи в наявності капітал, видає кредити.

Суть алгоритму полягає в наступному.

- Припустимо, що у системи в наявності n пристроїв, наприклад стрічок.
- ОС приймає запит від призначеного для користувача процесу, якщо його максимальна потреба не перевищує n.
- Користувач гарантує, що якщо ОС в змозі задовольнити його запит, то всі пристрої будуть повернені системі протягом кінцевого часу.
- Поточний стан системи називається надійним, якщо ОС може забезпечити всім процесам їх виконання протягом кінцевого часу.
- Відповідно до алгоритму банкіра виділення пристроїв можливе, тільки якщо стан системи залишається надійним.

Розглянемо приклад надійного стану для системи з 3 користувачами і 11 пристроями, де 9 пристроїв задіяно, а 2 є в резерві. Хай поточна ситуація така:

Користувачі	Максимальна необхідність в ресурсах	Виділена кількість ресурсів користувачам
Перший	9	6
Другий	10	2
Третій	3	1

Мал. 8.2. Приклад надійного стану для системи з 3 користувачами і 11 пристроями.

Даний стан надійний. Подальші дії системи можуть бути такі. Спочатку задовольнити запити третього користувача, потім дочекатися, коли він закінчить роботу і звільнить свої три пристрої. Потім можна обслужити першого і другого користувачів. Тобто система задовольняє тільки ті запити, які залишають її в надійному стані, і відхиляє останні.

Термін ненадійний стан не припускає, що обов'язково виникне безвихідь. Він лише говорить про те, що у разі несприятливої послідовності подій система може зайти в безвихідь.

Даний алгоритм володіє тією гідністю, що при його використанні немає необхідності в перерозподілі ресурсів і відкоті процесів назад. Проте використання цього методу вимагає виконання ряду умов.

- Число користувачів і число ресурсів фіксоване.
- Число працюючих користувачів повинне залишатися постійним.
- Алгоритм вимагає, щоб клієнти гарантовано повертали ресурси.
- Мають бути заздалегідь вказані максимальні вимоги процесів до ресурсів. Найчастіше дана інформація відсутня.

Наявність таких жорстких і часто неприйнятних вимог може схилити розробників до вибору інших вирішень проблеми взаємоблокування.

Запобігання тупикам за рахунок порушення умов виникнення безвиході

У відсутність інформації про майбутні запити єдиний спосіб уникнути взаємоблокування – добитися невиконання хоч би одного з умов розділу «Умови виникнення безвиході».

Порушення умови взаємовиключення

У загальному випадку уникнути тих, що взаємовиключають неможливо. Доступ до деяких ресурсів має бути винятковим. Проте деякі пристрої вдається усуспільнити. Як приклад розглянемо принтер. Відомо, що намагатися здійснювати вивід на принтер можуть декілька процесів. Щоб уникнути хаосу організовують проміжне формування всіх вихідних даних процесу на диску, тобто пристрої, що розділяється. Лише один системний процес, званий сервісом або демоном принтера, відповідає за виведення документів на друк у міру звільнення принтера, реально з ним взаємодіє. Ця схема називається спулінгом (spooling). Таким чином, принтер стає пристроєм, що розділяється, і безвихідь для нього усунена.

На жаль, не для всіх пристроїв і не для всіх даних можна організувати спулінг. Неприємним побічним наслідком такої моделі може бути потенційна тупикова ситуація із-за конкуренції за дисковий простір для буфера спулінга. Проте в тій або іншій формі ця ідея застосовується часто.

Порушення умови очікування додаткових ресурсів

Умови очікування ресурсів можна уникнути, зажадавши виконання стратегії двофазного захоплення.

- У першій фазі процес повинен запрошувати всі необхідні йому ресурси відразу. До тих пір, поки вони не надані, процес не може продовжувати виконання.
- Якщо в першій фазі деякі ресурси, які були потрібні даному процесору, вже зайняті іншими процесами, він звільняє всі ресурси, які були йому виділені, і намагається повторити першу фазу.

У відомому сенсі цей підхід нагадує вимога захоплення всіх ресурсів заздалегідь. Природно, що тільки спеціально організовані програми можуть бути припинені протягом першої фази і рестартовані згодом.

Таким чином, один із способів – змусити всі процеси зажадати потрібні ним ресурси перед виконанням («все або нічого»). Якщо система в змозі виділити процесу все необхідне, він може працювати до завершення. Якщо хоч би один з ресурсів зайнятий, процес чекатиме.

Дане рішення застосовується в пакетних мейнфреймах (mainframe), які вимагають від користувачів перерахувати всі необхідні його програмі ресурси. Іншим прикладом може служити механізм двофазної локалізації записів в СУБД. Проте в цілому подібний підхід не дуже привабливий і приводить до неефективного використання комп'ютера. Як вже наголошувалося, перелік майбутніх запитів до ресурсів рідко вдається спрогнозувати. Якщо така інформація є, то можна скористатися алгоритмом банкіра. Відмітимо також, що описуваний підхід суперечить парадигмі модульності в програмуванні, оскільки застосування повинне знати про передбачувані запити до ресурсів у всіх модулях.

Порушення принципу відсутності перерозподілу

Якби можна було відбирати ресурси у утримуючих їх процесів до завершення цих процесів, то вдалося б добитися невиконання третьої умови виникнення безвиході. Перерахуємо мінуси даного підходу.

По-перше, відбирати у процесів можна тільки ті ресурси, стан яких легко зберегти, а пізніше відновити, наприклад стан процесора. По-друге, якщо процес протягом деякого часу використовує певні ресурси, а потім звільняє ці ресурси, він може втратити результати роботи,

виконаної до справжнього моменту. Нарешті, наслідком даної схеми може бути дискримінація окремих процесів, у яких постійно відбирають ресурси.

Все питання в ціні подібного рішення, яка може бути дуже високою, якщо необхідність відбирати ресурси виникає часто.

Порушення умови кругового очікування

Важко запропонувати розумну стратегію, щоб уникнути останньої умови з розділу «Умови виникнення безвиході» – циклічного очікування.

Один із способів – упорядкувати ресурси. Наприклад, можна привласнити всім ресурсам унікальні номери і зажадати, щоб процеси запрошували ресурси в порядку їх зростання. Тоді кругове очікування виникнути не може. Після останнього запиту і звільнення всіх ресурсів можна дозволити процесу знову здійснити перший запит. Очевидно, що практично неможливо знайти порядок, який задовольнить всіх.

Один з небагатьох прикладів впорядковування ресурсів – створення ієрархії спін-блокувань в Windows 2000. Спін-блокування – простий спосіб синхронізації (питання синхронізації процесів розглянуті у відповідній лекції). Спін-блокування може бути захоплена і звільнена процесом. Класична тупикова ситуація виникає, коли процес P1 захоплює спін-блокування S1 і претендує на спін-блокування S2, а процес P2, захоплює спін-блокування S2 і хоче додатково захопити спін-блокування S1. Щоб цього уникнути, всі спін-блокировки поміщаються у впорядкований список. Захоплення може здійснюватися тільки в порядку, вказаному в списку.

Інший спосіб атаки умови кругового очікування – діяти відповідно до правила, згідно якому кожен процес може мати тільки один ресурс в кожен момент часу. Якщо потрібний другий ресурс – звільни перший. Очевидно, що для багатьох процесів це неприйнятно.

Таким чином, технологія запобігання циклічному очікуванню, як правило, неефективна і може без необхідності закривати доступ до ресурсів.

Виявлення тупиків

Виявлення взаємоблокування зводиться до фіксації тупикової ситуації і виявлення залучених в неї процесів. Для цього проводиться перевірка наявності циклічного очікування у випадках, коли виконано перші три умови виникнення безвиході. Методи виявлення активно використовують графи розподілу ресурсів.

Розглянемо модельну ситуацію.

- Процес P1 чекає ресурс R1.
- Процес P2 утримує ресурс R2 і чекає ресурс R1.
- Процес P3 утримує ресурс R1 і чекає ресурс R3.
- Процес P4 чекає ресурс R2.
- Процес P5 утримує ресурс R3 і чекає ресурс R2.

Питання полягає в тому, чи є дана ситуація тупиковою, і якщо так, то які процеси в ній беруть участь. Для відповіді на це питання можна сконструювати граф ресурсів, як показано на мал. 8.2. З малюнка видно, що є цикл, що моделює умову кругового очікування, і що процеси P2,P3,P5, а може бути, та інші знаходяться в тупиковій ситуації.

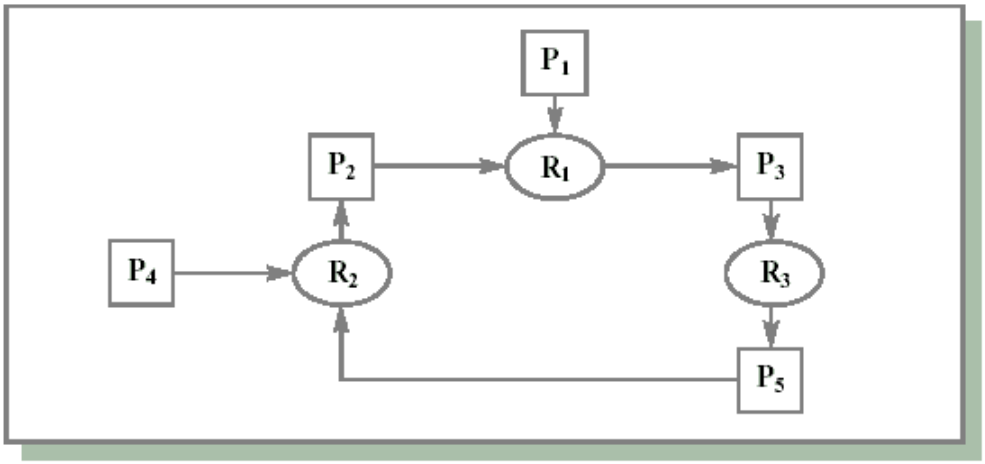


Рис. 8.3. Граф ресурсів

Візуально легко виявити наявність безвиході, але потрібні також формальні алгоритми, що реалізовуються на комп'ютері.

Один з таких алгоритмів описаний в [Таненбаум, 2002], там же можна знайти посилання на інші алгоритми.

Існують і інші способи виявлення безвиході, застосовні також в ситуаціях, коли є декілька ресурсів кожного типу. Так в [Дейтел, 1987] описаний спосіб, званий редукцією графа розподілу ресурсів, а в [Таненбаум, 2002] – матричний алгоритм.

Відновлення після тупиків

Виявивши безвихідь, можна вивести з нього систему, порушивши одну з умов існування безвиході. При цьому, можливо, декілька процесів часткові або повністю втратять результати виконаної роботи.

Складність відновлення обумовлена рядом чинників.

- У більшості систем немає достатньо ефективних засобів, щоб припинити процес, вивести його з системи і відновити згодом з того місця, де він був зупинений.
- Якщо навіть такі засоби є, то їх використання вимагає витрат і уваги оператора.
- Відновлення після безвиході може зажадати значних зусиль.

Найпростіший і найбільш поширений спосіб усунути безвихідь – завершити виконання одне або більш за процеси, щоб згодом використовувати його ресурси. Тоді у разі успіху решта процесів зможе виконуватися. Якщо це не допомагає, можна ліквідовувати ще декілька процесів. Після кожної ліквідації повинен запускатися алгоритм виявлення безвиході.

По можливості краще ліквідовувати той процес, який може бути без збитку повернений до початку (такі процеси називаються ідемпотентними). Прикладом такого процесу може служити компіляція. З іншого боку, процес, який змінює вміст бази даних, не завжди може бути коректно запущений повторно.

В деяких випадках можна тимчасово забрати ресурс у поточного власника і передати його іншому процесу. Можливість забрати ресурс у процесу, дати його іншому процесу і потім без збитку повернути назад сильно залежить від природи ресурсу. Подібне відновлення часто скрутно, якщо не неможливо.

У ряді систем реалізовані засоби відкоту і перезапуску або рестарту з контрольної крапки (збереження стану системи в якийсь момент часу). Якщо проектувальники системи знають, що безвихідь вірогідна, вони можуть періодично організовувати для процесів контрольні крапки. Іноді це доводиться робити розробникам прикладних програм.

Коли безвихідь виявлена, видно, які ресурси залучені в цикл кругового очікування. Щоб здійснити відновлення, процес, який володіє таким ресурсом, має бути відкинутий до моменту часу, передуванню його запиту на цей ресурс.

Висновок

Виникнення безвиході є потенційною проблемою будь-якої операційної системи. Вони виникають, коли є група процесів, кожен з яких намагається дістати винятковий доступ до деяких ресурсів і претендує на ресурси, що належать іншому процесу. У результаті всі вони виявляються в стані нескінченного очікування.

З тупиками можна боротися, можна їх виявляти, уникати і відновлювати систему після тупиків. Проте ціна подібних дій висока і відповідні зусилля повинні робитися тільки в системах, де ігнорування тупикових ситуацій приводить до катастрофічних наслідків.

Модуль 3. Управління пам’яттю в операційних системах.

Введення

Головне завдання комп'ютерної системи – виконувати програми. Програми разом з даними, до яких вони мають доступ, в процесі виконання повинні (принаймні частково) знаходитися в оперативній пам'яті. Операційній системі доводиться вирішувати задачу розподілу пам'яті між призначеними для користувача процесами і компонентами ОС. Ця діяльність називається управлінням пам'яттю. Таким чином, пам'ять (storage, memory) є найважливішим ресурсом, що вимагає ретельного управління. У недавньому минулому пам'ять була найдорожчим ресурсом.

Частина ОС, яка відповідає за управління пам'яттю, називається менеджером пам'яті.

Фізична організація пам'яті комп'ютера

Пристрої комп'ютера, що запам'ятовують, розділяють, як мінімум, на два рівні: основну (головну, оперативну, фізичну) і вторинну (зовнішню) пам'ять.

Основна пам'ять є впорядкованим масивом однобайтових осередків, кожна з яких має свою унікальну адресу (номер). Процесор витягує команду з основної пам'яті, декодує і виконує її. Для виконання команди можуть потрібно звернення ще до декількох елементів основної пам'яті. Зазвичай основна пам'ять виготовляється із застосуванням напівпровідникових технологій і втрачає свій вміст при відключенні живлення.

Вторинну пам'ять (це головним чином диски) також можна розглядати як одновимірний лінійний адресний простір, що складається з послідовності байтів. На відміну від оперативної пам'яті, вона є незалежною, має істотно велику ємкість і використовується як розширення основної пам'яті.

Цю схему можна доповнити ще декількома проміжними рівнями, як показано на мал. 9.1. Різновиди пам'яті можуть бути об'єднані в ієрархію по убуванню часу доступу, зростанню ціни і збільшенню ємкості.

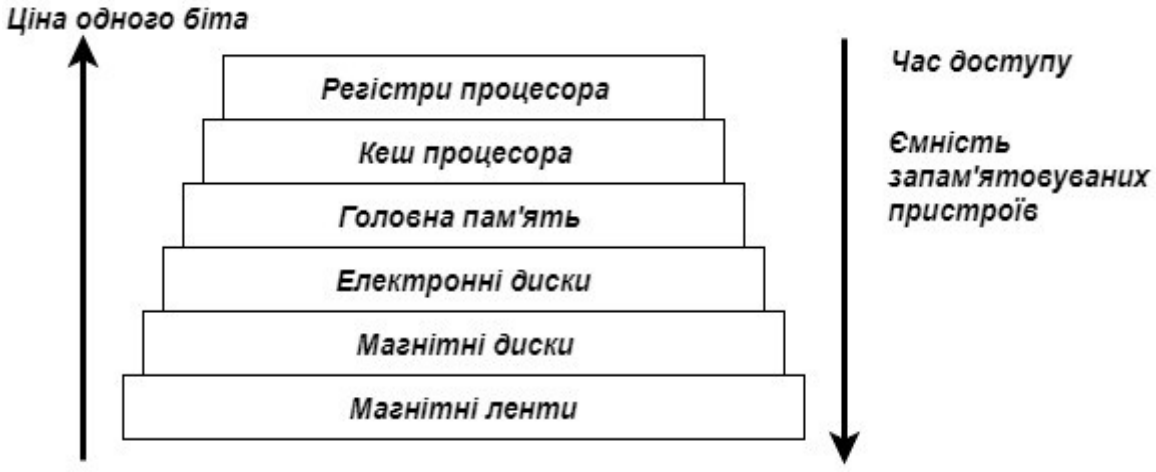


Рис. 9.1. Ієрархія пам'яті

Багаторівневу схему використовують таким чином. Інформація, яка знаходиться в пам'яті верхнього рівня, зазвичай зберігається також на рівнях з великими номерами. Якщо процесор не виявляє потрібну інформацію на і-м рівні, він починає шукати її на наступних рівнях. Коли потрібна інформація знайдена, вона переноситься в швидші рівні.

Локальність

Виявляється, при такому способі організації у міру зниження швидкості доступу до рівня пам'яті знижується також і частота звернень до нього.

Ключову роль тут грає властивість реальних програм, протягом обмеженого відрізання часу здатних працювати з невеликим набором адрес пам'яті. Це емпірично спостережувана властивість відома як принцип локальності або локалізації звернень.

Властивість локальності (сусідні у просторі та часі об'єкти характеризуються схожими властивостями) властиво не тільки функціонуванню ОС, але і природі взагалі. У разі ОС властивість локальності з'ясовна, якщо врахувати, як пишуться програми і як зберігаються дані, тобто зазвичай протягом якогось відрізання часу обмежений фрагмент коди працює з обмеженим набором даних. Цю частину коди і даних вдається розмістити в пам'яті з швидким доступом. В результаті реальний час доступу до пам'яті визначається часом доступу до верхніх рівнів, що і обумовлює ефективність використання ієрархічної схеми. Треба сказати, що описувана організація обчислювальної системи багато в чому імітує діяльність людського мозку при переробці інформації. Дійсно, вирішуючи конкретну проблему, людина працює з невеликим об'ємом інформації, зберігаючи відомості, що не відносяться до справи, в своїй пам'яті або в зовнішній пам'яті (наприклад, в книгах).

Кеш процесора зазвичай є частиною апаратури, тому менеджер пам'яті ОС займається розподілом інформації головним чином в основній і зовнішній пам'яті комп'ютера. У деяких схемах потоки між оперативною і зовнішньою пам'яттю регулюються програмістом (див. наприклад, далі оверлейні структури), проте це пов'язано з витратами часу програміста, так що подібну діяльність прагнуть покласти на ОС.

Адреси в основній пам'яті, що характеризують реальне розташування даних у фізичній пам'яті, називаються фізичними адресами. Набір фізичних адрес, з яким працює програма, називають фізичним адресним простором.

Логічна пам'ять

Апаратна організація пам'яті у вигляді лінійного набору осередків не відповідає уявленням програміста про те, як організовано зберігання програм і даних. Більшістю програм є набір модулів, створених незалежно один від одного. Іноді всі модулі, що входять до складу процесу, розташовуються в пам'яті один за іншим, утворюючи лінійний простір адрес. Проте частіше модулі поміщаються в різні області пам'яті і використовуються по-різному.

Схема управління пам'яттю, що підтримує цей погляд користувача на те, як зберігаються програми і дані, називається сегментацією. Сегмент – область пам'яті певного призначення, усередині якої підтримується лінійна адресація. Сегменти містять процедури, масиви, стік або скалярні величини, але зазвичай не містять інформацію змішаного типу.

Мабуть, спочатку сегменти пам'яті з'явилися у зв'язку з необхідністю усуспільнення процесами фрагментів програмної коди (текстовий редактор, тригонометричні бібліотеки і т. д.), без чого кожен процес повинен був зберігати в своєму адресному просторі дублюючу інформацію. Ці окремі ділянки пам'яті, що зберігають інформацію, яку система відображає в пам'ять декількох процесів, отримали назву сегментів. Пам'ять, таким чином, перестала бути лінійною і перетворилася на двовимірну. Адреса складається з двох компонентів: номер сегменту, зсув усередині сегменту. Далі виявилося зручним розміщувати в різних сегментах різні компоненти процесу (код програми, дані, стек і т. д.). Попутно з'ясувалося, що можна контролювати характер роботи з конкретним сегментом, приписавши йому атрибути, наприклад права доступу або типи операцій, які дозволяється проводити з даними, що зберігаються в сегменті.

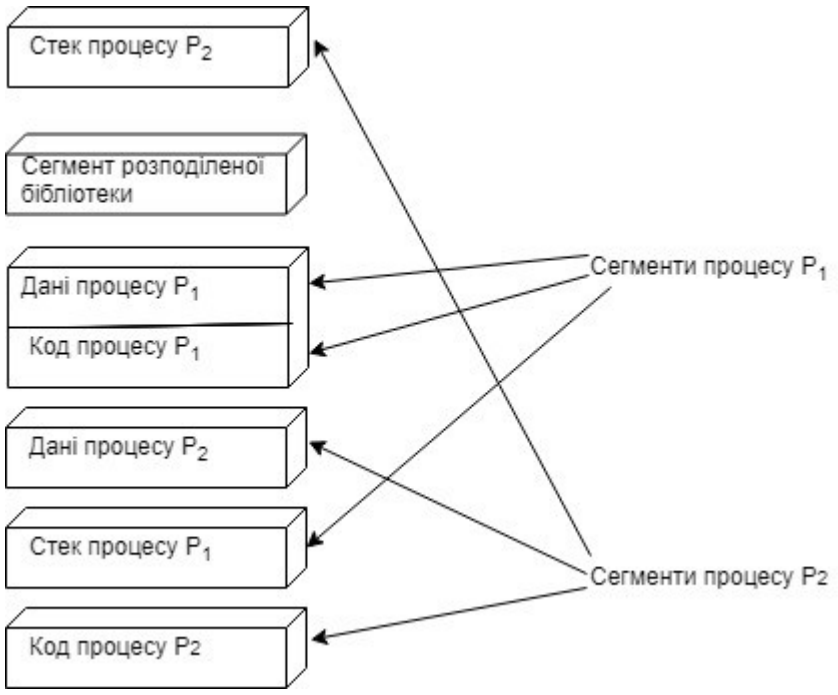


Рис. 9.2. Розташування сегментів процесів в пам'яті комп'ютера

Деякі сегменти, що описують адресний простір процесу, показані на мал. 9.2.

Більшість сучасних ОС підтримують сегментну організацію пам'яті. У деякій архітектурі (Intel, наприклад) сегментація підтримується устаткуванням.

Адреси, до яких звертається процес, таким чином, відрізняються від адрес, що реально існують в оперативній пам'яті. У кожному конкретному випадку використовувані програмою адреси можуть бути представлені різними способами. Наприклад, адреси в початкових текстах зазвичай символічні. Компілятор пов'язує ці символічні адреси з переміщуваними адресами (такими, як n байт від початку модуля). Подібна адреса, що згенерувала програмою, зазвичай називають логічною (у системах з віртуальною пам'яттю він часто називається віртуальним) адресою. Сукупність всіх логічних адрес називається логічним (віртуальним) адресним простором.

Скріплення адрес

Отже логічні і фізичні адресні простори ні по організації, ні за розміром не відповідають один одному. Максимальний розмір логічного адресного простору зазвичай визначається розрядністю процесора (наприклад, 232) і в сучасних системах значно перевищує розмір фізичного адресного простору. Отже, процесор і ОС мають бути здатні відобразити посилання в коді програми в реальні фізичні адреси, відповідні поточному розташуванню програми в основній пам'яті. Таке відображення адрес називають трансляцією (прив'язкою) адреси або скріпленням адрес (див. мал. 9.3).

Скріплення логічної адреси, породженої оператором програми, з фізичним повинно бути здійснено до початку виконання оператора або у момент його виконання. Таким чином, прив'язка інструкцій і даних до пам'яті в принципі може бути зроблена на наступних кроках [Silberschatz, 2002].

- Етап компіляції (Compile time). Коли на стадії компіляції відоме точне місце розміщення процесу в пам'яті, тоді безпосередньо генеруються фізичні адреси. При зміні стартової

адреси програми необхідно перекомпілювати її код. Як приклад можна привести .com програми MS-DOS, які пов'язують її з фізичними адресами на стадії компіляції.

- Етап завантаження (Load time). Якщо інформація про розміщення програми на стадії компіляції відсутня, компілятор генерує переміщуваний код. В цьому випадку остаточне скріплення відкладається до моменту завантаження. Якщо стартова адреса міняється, потрібно всього лише перезавантажити код з урахуванням зміненої величини.
- Етап виконання (Execution time). Якщо процес може бути переміщений під час виконання з однієї області пам'яті в іншу, скріплення відкладається до стадії виконання. Тут бажана наявність спеціалізованого устаткування, наприклад регістрів переміщення. Їх значення додається до кожної адреси, що згенерувала процесом. Більшість сучасних ОС здійснюють трансляцію адрес на етапі виконання, використовуючи для цього спеціальний апаратний механізм.

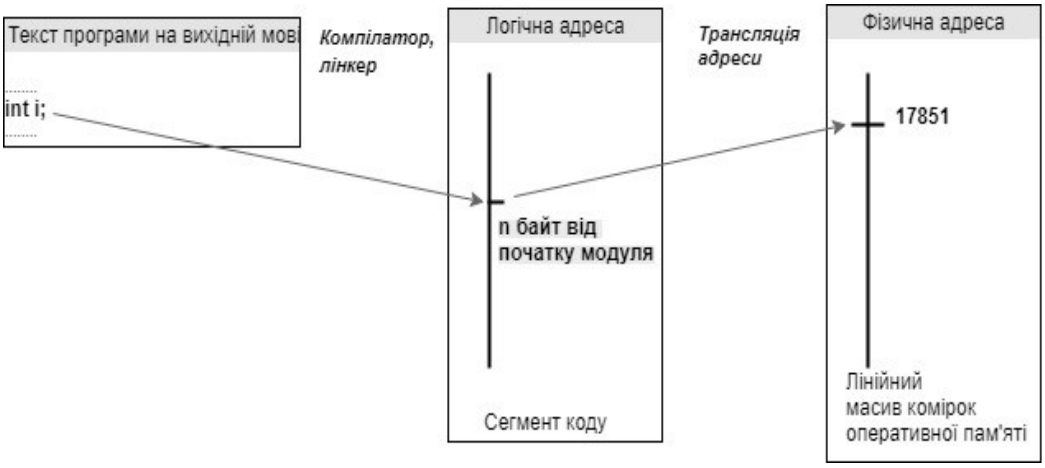


Рис. 9.3. Формування логічної адреси і скріплення логічної адреси з фізичним

Функції системи управління пам'яттю

Щоб забезпечити ефективний контроль використання пам'яті, ОС повинна виконувати наступні функції:

- відображення адресного простору процесу на конкретні області фізичної пам'яті;
- розподіл пам'яті між конкуруючими процесами;
- контроль доступу до адресних просторів процесів;
- вивантаження процесів (цілком або частково) в зовнішню пам'ять, коли в оперативній пам'яті недостатньо місця;
- облік вільної і зайнятої пам'яті.

У наступних розділах лекції розглядається ряд конкретних схем управління пам'яттю. Кожна схема включає певну ідеологію управління, а також алгоритми і структури даних і залежить від архітектурних особливостей використовуваної системи. Спочатку будуть розглянуті прості схеми. Домінуюча на сьогодні схема віртуальної пам'яті буде описана в подальших лекціях.

Прості схеми управління пам'яттю

Перші ОС застосовували дуже прості методи управління пам'яттю. Спочатку кожен процес користувача повинен був повністю поміститися в основній пам'яті, займати безперервну область пам'яті, а система приймала до обслуговування додаткові призначені для користувача процеси до тих пір, поки всі вони одночасно поміщалися в основній пам'яті. Потім з'явився "простий свопінг" (система як і раніше розміщує кожен процес в основній пам'яті цілком, але іноді на підставі деякого критерію цілком скидає образ деякого процесу з основної пам'яті в зовнішню і замінює його в основній пам'яті образом іншого процесу). Такого роду схеми мають не тільки історичну цінність. В даний час вони застосовуються в учбових і науково-дослідних модельних ОС, а також в ОС для вбудованих (embedded) комп'ютерів.

Схема з фіксованими розділами

Найпростішим способом управління оперативною пам'яттю є її попереднє (зазвичай на етапі генерації або у момент завантаження системи) розбиття на декілька розділів фіксованої величини. Процеси, що поступають, поміщаються в той або інший розділ. При цьому відбувається умовне розбиття фізичного адресного простору. Скріплення логічних і фізичних адрес процесу відбувається на етапі його завантаження в конкретний розділ, іноді – на етапі компіляції.

Кожен розділ може мати свою чергу процесів, а може існувати і глобальна черга для всіх розділів(див. мал. 9.4).

Ця схема була реалізована в IBM OS/360 (MFT), DEC RSX-11 і ряду інших систем.

Підсистема управління пам'яттю оцінює розмір процесу, що поступив, вибирає відповідний для нього розділ, здійснює завантаження процесу в цей розділ і налаштування адрес.

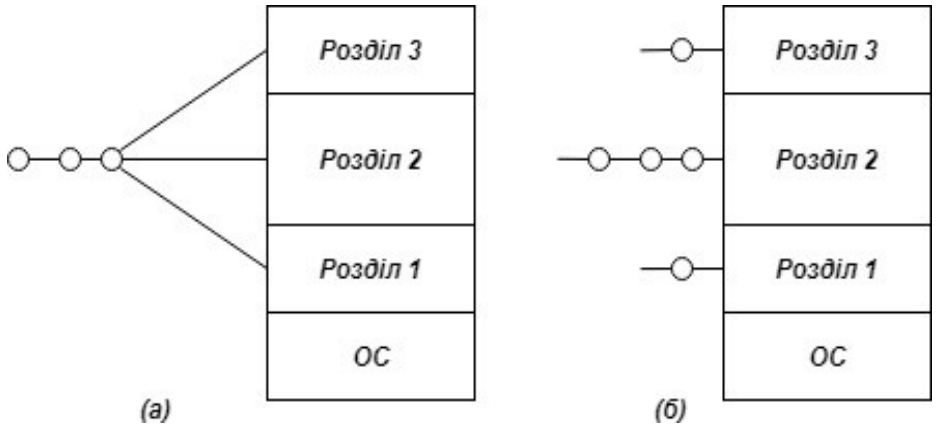


Рис. 9.4. Схема з фіксованими розділами: (a) – із загальною чергою процесів, (b) – з окремими чергами процесів

Очевидний недолік цієї схеми – число одночасно виконуваних процесів обмежене числом розділів.

Іншим істотним недоліком є те, що пропонована схема сильно страждає від внутрішньої фрагментації – втрати частини пам'яті, виділеної процесу, але не використовуваною їм. Фрагментація виникає тому, що процес не повністю займає виділений йому розділ або тому, що деякі розділи дуже малі для виконуваних призначених для користувача програм.

Один процес в пам'яті

Окремий випадок схеми з фіксованими розділами – робота менеджера пам'яті однозадачної ОС. У пам'яті розміщується один призначений для користувача процес. Залишається визначити, де розташовується призначена для користувача програма по відношенню до ОС – у верхній частині пам'яті, в нижній або в середній. Причому частина ОС може бути в ROM (наприклад, BIOS, драйвери пристроїв). Головний чинник, що впливає на це рішення, – розташування вектора переривань, який зазвичай локалізований в нижній частині пам'яті, тому ОС також розміщують в нижній. Прикладом такої організації може служити ОС MS-DOS.

Захист адресного простору ОС від призначеної для користувача програми може бути організована за допомогою одного граничного регістра, що містить адресу межі ОС.

Оверлейна структура

Оскільки розмір логічного адресного простору процесу може бути більше, ніж розмір виділеного йому розділу (або більше, ніж розмір найбільшого розділу), іноді використовується техніка, звана оверлей (overlay) або організація структури з перекриттям. Основна ідея – тримати в пам'яті тільки ті інструкції програми, які потрібні в даний момент.

Потреба в такому способі завантаження з'являється, якщо логічний адресний простір системи малий, наприклад 1 Мбайт (MS-DOS) або навіть всього 64 Кбайта (PDP-11), а програма відносно велика. На сучасних 32-розрядних системах, де віртуальний адресний простір вимірюється гігабайтами, проблеми з браком пам'яті вирішуються іншими способами (див. розділ "Віртуальна пам'ять").

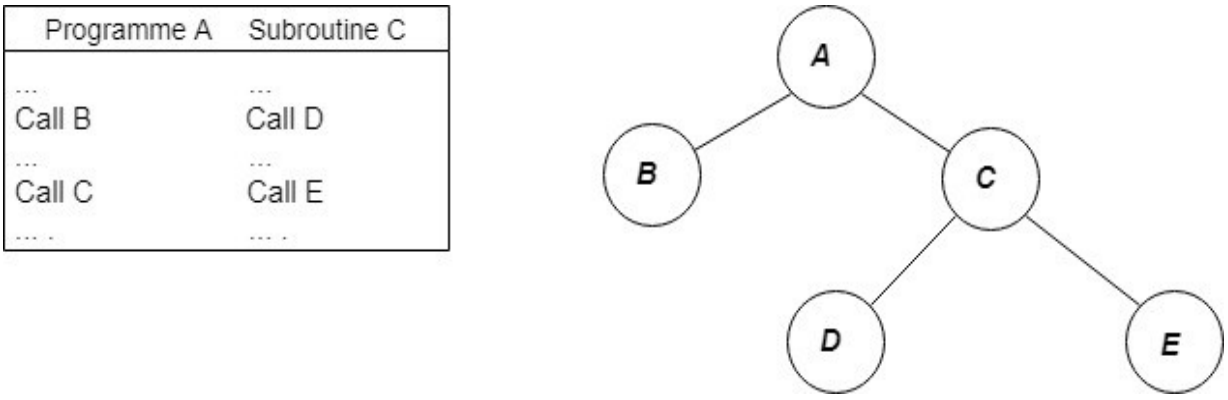


Рис. 9.5. Організація структури з перекриттям. Можна по черзі завантажувати в пам'ять гілки A-B, A-C-D і A-C-E програми

Коди гілок оверлейної структури програми знаходяться на диску як абсолютні образи пам'яті і прочитуються драйвером оверлеїв при необхідності. Для опису оверлейної структури зазвичай використовується спеціальна нескладна мова (overlay description language). Сукупність файлів виконуваної програми доповнюється файлом (зазвичай з розширенням .odl), що описує дерево викликів усередині програми. Для прикладу, приведеного на мал. 9.5, текст цього файлу може виглядати так:

A-(B,C)

C-(D,E)

Синтаксис подібного файлу може розпізнаватися завантажувачем. Прив'язка до фізичної пам'яті відбувається у момент чергового завантаження одного з гілок програми.

Оверлеї можуть бути повністю реалізовані на призначеному для користувача рівні в системах з простий файловою структурою. ОС при цьому лише робить дещо більше операцій введення-виводу. Типове рішення – породження лінкером спеціальних команд, які включають завантажувач кожного разу, коли потрібне звернення до однієї з гілок програми, що перекриваються.

Ретельне проектування оверлейної структури віднімає багато часу і вимагає знання пристрою програми, її коди, даних і мови опису оверлейної структури. З цієї причини застосування оверлеїв обмежене комп'ютерами з невеликим логічним адресним простором. Як ми побачимо надалі, проблема оверлейних сегментів, контрольованих програмістом, відпадає завдяки появі систем віртуальної пам'яті.

Відмітимо, що можливість організації структур з перекриттями багато в чому обумовлена властивістю локальності, яка дозволяє зберігати в пам'яті тільки ту інформацію, яка необхідна в конкретний момент обчислень.

Динамічний розподіл. Свопінг

Маючи справу з пакетними системами, можна обходитися фіксованими розділами і не використовувати нічого складнішого. У системах з розділенням часу можлива ситуація, коли пам'ять не в змозі містити всі призначені для користувача процеси. Доводиться удаватися до свопінгу (swapping) – переміщення процесів з головної пам'яті на диск і назад цілком. Часткове вивантаження процесів на диск здійснюється в системах із сторінковою організацією (paging) і буде розглянута нижче.

Вивантажений процес може бути повернений в той же самий адресний простір або в інше. Це обмеження диктується методом скріплення. Для схеми скріплення на етапі виконання можна завантажити процес в інше місце пам'яті.

Свопінг не має безпосереднього відношення до управління пам'яттю, швидше він пов'язаний з підсистемою планування процесів. Очевидно, що свопінг збільшує час перемикання контексту. Час вивантаження може бути скорочене за рахунок організації спеціально відведеного простору на диску (розділ для свопінгу). Обмін з диском при цьому здійснюється блоками більшого розміру, тобто швидше, ніж через стандартну файлову систему. У багатьох версіях Unix свопінг починає працювати тільки тоді, коли виникає необхідність в зниженні завантаження системи.

Схема із змінними розділами

В принципі, система свопінгу може базуватися на фіксованих розділах. Ефективнішою, проте, представляється схема динамічного розподілу або схема із змінними розділами, яка може використовуватися і в тих випадках, коли всі процеси цілком поміщаються в пам'яті, тобто у відсутність свопінгу. В цьому випадку спочатку вся пам'ять вільна і не розділена заздалегідь на розділи. Завданню, що знов поступає, виділяється строго необхідна кількість пам'яті, не більш. Після вивантаження процесу пам'ять тимчасово звільняється. Після закінчення деякого часу пам'яттю є змінне число розділів різного розміру (мал. 10.1). Суміжні вільні ділянки можуть бути об'єднані.

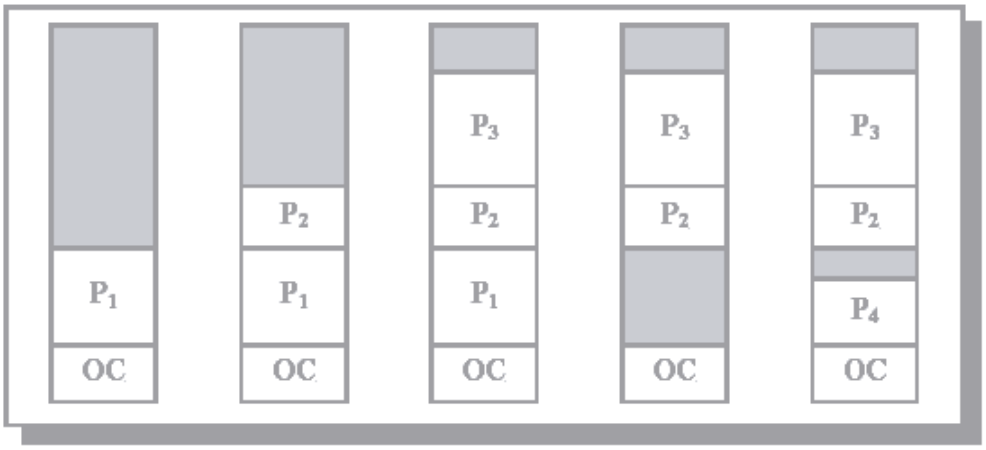


Рис. 10.1. Динаміка розподілу пам'яті між процесами (сірим кольором показана невживана пам'ять)

У який розділ поміщати процес? Найбільш поширено три стратегії.

- Стратегія першого відповідного (First fit). Процес поміщається в перший відповідний за розміром розділ.
- Стратегія найбільш відповідного (Best fit). Процес поміщається в той розділ, де після його завантаження залишиться менше всього вільного місця.
- Стратегія найменш відповідного (Worst fit). При приміщенні в найбільший розділ в нім залишається достатньо місця для можливого розміщення ще одного процесу.

Моделювання показало, що частка корисно використовуваної пам'яті по-перше двох випадках більше, при цьому перший спосіб декілька швидше. Попутно відмітимо, що перераховані стратегії широко застосовуються і іншими компонентами ОС, наприклад для розміщення файлів на диску.

Типовий цикл роботи менеджера пам'яті полягає в аналізі запиту на виділення вільної ділянки (розділу), виборі його серед тих, що є відповідно до однієї із стратегій (першого відповідного, найбільш відповідного і найменш відповідного), завантаженні процесу у вибраний розділ і подальших змінах таблиць вільних і зайнятих областей. Аналогічне коректування необхідне і після завершення процесу. Скріплення адрес може здійснюватися на етапах завантаження і виконання.

Цей метод гнучкіший в порівнянні з методом фіксованих розділів, проте йому властива зовнішня фрагментація – наявність великого числа ділянок невживаної пам'яті, не виділеної жодному процесу. Вибір стратегії розміщення процесу між першим відповідним і найбільш

відповідним слабо впливає на величину фрагментації. Цікаво, що метод найбільш відповідного може виявитися найгіршим, оскільки він залишає безліч дрібних незайнятих блоків.

Статистичний аналіз показує, що пропадає в середньому 1/3 пам'яті! Це відоме правило 50% (дві сусідні вільні ділянки на відміну від двох сусідніх процесів можуть бути об'єднані).

Одне з вирішень проблеми зовнішньої фрагментації – організувати стискування, тобто переміщення всіх зайнятих (вільних) ділянок у бік зростання (убування) адрес, так, щоб вся вільна пам'ять утворила безперервну область. Цей метод іноді називають схемою з переміщуваними розділами. У ідеалі фрагментація після стискування має бути відсутньою. Стискування, проте, є дорогою процедурою, алгоритм вибору оптимальної стратегії стискування дуже важкий і, як правило, стискування здійснюється в комбінації з вивантаженням і завантаженням по інших адресах.

Сторінкова пам'ять

Описані вище схеми недостатньо ефективно використовують пам'ять, тому в сучасних схемах управління пам'яттю не прийнято розміщувати процес в оперативній пам'яті одним безперервним блоком.

У найпростішому і найбільш поширеному випадку сторінкової організації пам'яті (або paging) як логічний адресний простір, так і фізичне представляються такими, що складаються з наборів блоків або сторінок однакового розміру. При цьому утворюються логічні сторінки (page), а відповідні одиниці у фізичній пам'яті називають фізичними сторінками або сторінковими кадрами (page frames). Сторінки (і сторінкові кадри) мають фіксовану довжину, що зазвичай є ступенем числа 2, і не можуть перекриватися. Кожен кадр містить одну сторінку даних. При такій організації зовнішня фрагментація відсутня, а втрати через внутрішню фрагментацію, оскільки процес займає ціле число сторінок, обмежені частиною останньої сторінки процесу.

Логічна адреса в сторінковій системі – впорядкована пара (p,d), де p – номер сторінки у віртуальній пам'яті, а d – зсув в рамках сторінки p, на якій розміщується елемент, що адресується. Відмітимо, що розбиття адресного простору на сторінки здійснюється обчислювальною системою непомітно для програміста. Тому адреса є двовимірною лише з погляду операційної системи, а з погляду програміста адресний простір процесу залишається лінійним.

Описувана схема дозволяє завантажити процес, навіть якщо немає безперервної області кадрів, достатньої для розміщення процесу цілком. Але одного базового регістра для здійснення трансляції адреси в даній схемі недостатньо. Система відображення логічних адрес у фізичних зводиться до системи відображення логічних сторінок у фізичних і є таблицею сторінок, яка зберігається в оперативній пам'яті. Іноді говорять, що таблиця сторінок – це кусково-лінійна функція відображення, задана в табличному вигляді.

Інтерпретація логічної адреси показана на мал. 10.2. Якщо виконуваний процес звертається до логічної адреси v = (p,d), механізм відображення шукає номер сторінки p в таблиці сторінок і визначає, що ця сторінка знаходиться в сторінковому кадрі p', формуючи реальну адресу з p' і d.

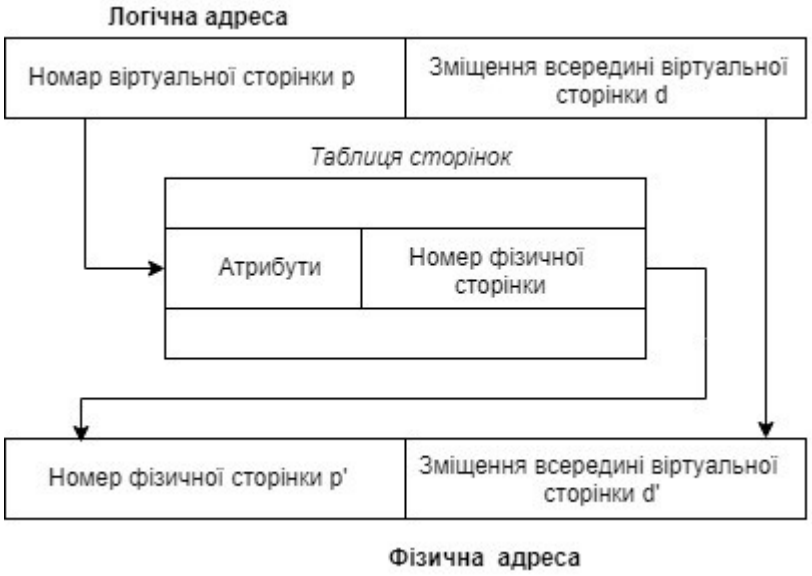


Рис. 10.2. Зв'язок логічної і фізичної адрес при сторінковій організації пам'яті

Таблиця сторінок (page table) адресується за допомогою спеціального регістра процесора і дозволяє визначити номер кадру за логічною адресою. Окрім цього основного завдання, за допомогою атрибутів, записаних в рядку таблиці сторінок, можна організувати контроль доступу до конкретної сторінки і її захист.

Відзначимо ще раз відмінність точок зору користувача і системи на використовувану пам'ять. З погляду користувача, його пам'ять – єдиний безперервний простір, що містить тільки одну програму. Реальне відображення приховане від користувача і контролюється ОС. Відмітимо, що процесу користувача чужа пам'ять недоступна. Він не має можливості адресувати пам'ять за межами своєї таблиці сторінок, яка включає тільки його власні сторінки.

Для управління фізичною пам'яттю ОС підтримує структуру таблиці кадрів. Вона має один запис на кожен фізичний кадр, що показує його стан.

Відображення адрес має бути здійснене коректно навіть в складних випадках і зазвичай реалізується апаратно. Для посилання на таблицю процесів використовується спеціальний регістр. При перемиканні процесів необхідно знайти таблицю сторінок нового процесу, покажчик на яку входить в контекст процесу.

Сегментна і сегментно-сторінкова організація пам'яті

Існують дві інші схеми організації управління пам'яттю: сегментна і сегментно-сторінкова. Сегменти, на відміну від сторінок, можуть мати змінний розмір. Ідея сегментації викладена у вступі. При сегментній організації віртуальна адреса є двовимірною як для програміста, так і для операційної системи, і складається з двох полів – номери сегменту і зсуву усередині сегменту. Підкреслимо, що на відміну від сторінкової організації, де лінійна адреса перетворена в двовимірний операційною системою для зручності відображення, тут двовимірна адреси є наслідком представлення користувача про процес не у вигляді лінійного масиву байтів, а як набір сегментів змінного розміру (дані, код, стек...).

Програмісти, що пишуть на мовах низького рівня, повинні мати уявлення про сегментну організацію, явним чином міняючи значення сегментних регістрів (це добре видно по текстах програм, написаних на Асемблері). Логічний адресний простір – набір сегментів. Кожен сегмент

має ім'я, розмір і інші параметри (рівень привілеїв, дозволені види звернень, прапори присутності). На відміну від сторінкової схеми, де користувач задає тільки один адресу, яка розбивається на номер сторінки і зсув прозорим для програміста чином, в сегментній схемі користувач специфікує кожну адресу двома величинами: ім'ям сегменту і зсувом.

Кожен сегмент – лінійна послідовність адрес, що починається з 0. Максимальний розмір сегменту визначається розрядністю процесора (при 32-розрядній адресації це 232 байт або 4 Гбайт). Розмір сегменту може мінятися динамічно (наприклад, сегмент стека). У елементі таблиці сегментів окрім фізичної адреси початку сегменту зазвичай міститься і довжина сегменту. Якщо розмір зсуву у віртуальній адресі виходить за межі розміру сегменту, виникає виняткова ситуація.

Логічна адреса – впорядкована пара v=(s,d), номер сегменту і зсув усередині сегменту.

У системах, де сегменти підтримуються апаратно, ці параметри зазвичай зберігаються в таблиці дескрипторів сегментів, а програма звертається до цих дескрипторів по номерах-селекторах. При цьому в контекст кожного процесу входить набір сегментних регістрів, селектори поточних сегментів коди, що містять, стеки, дані і так далі і що визначають, які сегменти використовуватимуться при різних видах звернень до пам'яті. Це дозволяє процесору вже на апаратному рівні визначати допустимість звернень до пам'яті, спрощуючи реалізацію захисту інформації від пошкодження і несанкціонованого доступу.

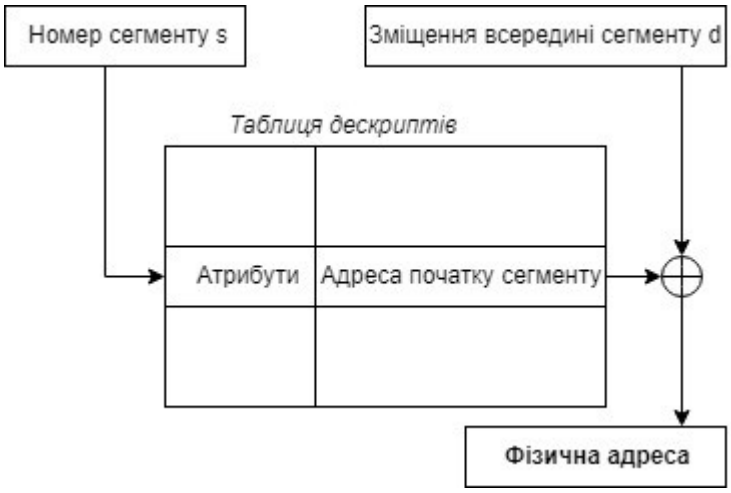


Рис. 10.3. Перетворення логічної адреси при сегментній організації пам'яті

Апаратна підтримка сегментів поширена мало (головним чином на процесорах Intel). У більшості ОС сегментація реалізується на рівні, не залежному від апаратури.

Зберігати в пам'яті сегменти великого розміру цілком так само незручно, як і зберігати процес безперервним блоком. Напрошується ідея розбиття сегментів на сторінки. При сегментно-сторінковій організації пам'яті відбувається дворівнева трансляція віртуальної адреси у фізичну. В цьому випадку логічна адреса складається з трьох полів: номери сегменту логічної пам'яті, номери сторінки усередині сегменту і зсуву усередині сторінки. Відповідно, використовуються дві таблиці відображення – таблиця сегментів, що зв'язує номер сегменту з таблицею сторінок, і окрема таблиця сторінок для кожного сегменту.

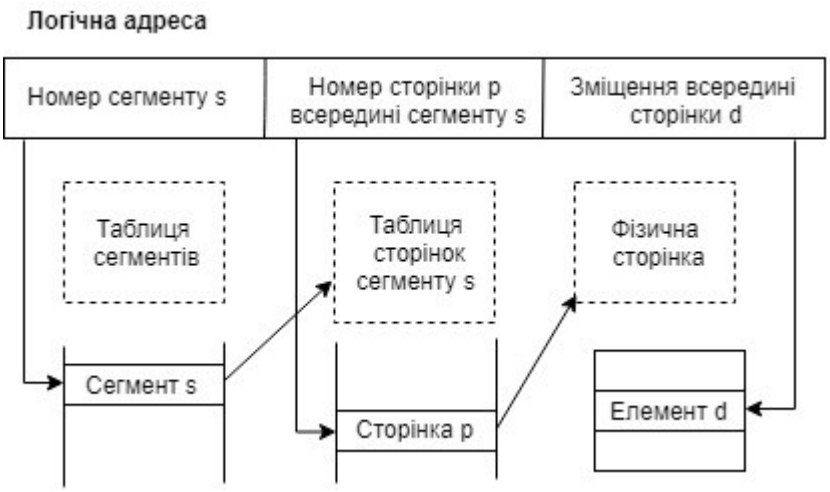


Рис. 10.4. Спрощена схема формування фізичної адреси при сегментно-сторінковій організації пам'яті

Сегментно-сторінкова і сторінкова організація пам'яті дозволяє легко організувати сумісне використання одних і тих же даних і програмної коди різними завданнями. Для цього різні логічні блоки пам'яті різних процесів відображають в один і той же блок фізичної пам'яті, де розміщується фрагмент коди або даних, що розділяється.

Висновок

У лекції описані прості способи управління пам'яттю в ОС. Фізична пам'ять комп'ютера має ієрархічну структуру. Програма є набором сегментів в логічному адресному просторі. ОС здійснює скріплення логічних і фізичних адресних просторів. У подальших лекціях розглядатимуться сучасні рішення, пов'язані з підтримкою віртуальної пам'яті.

Поняття віртуальної пам'яті

Розробникам програмного забезпечення часто доводиться вирішувати проблему розміщення в пам'яті великих програм, розмір яких перевищує об'єм доступної оперативної пам'яті. Один з варіантів вирішення даної проблеми – організація структур з перекриттям – розглянутий в попередній лекції. При цьому передбачалася активна участь програміста в процесі формування частин програми, що перекриваються. Розвиток архітектури комп'ютерів і розширення можливостей операційної системи по управлінню пам'яттю дозволив перекласти рішення цієї задачі на комп'ютер. Одним з головних досягнень стала поява віртуальної пам'яті (virtual memory). Вперше вона була реалізована в 1959 р. на комп'ютері «Атлас», розробленому в Манчестерському університеті.

Суть концепції віртуальної пам'яті полягає в наступному. Інформація, з якою працює активний процес, повинна розташовуватися в оперативній пам'яті. У схемах віртуальної пам'яті у процесу створюється ілюзія того, що вся необхідна йому інформація є в основній пам'яті. Для цього, по-перше, займана процесом пам'ять розбивається на декілька частин, наприклад сторінок. По-друге, логічна адреса (логічна сторінка), до якої звертається процес, динамічно транслюється у фізичну адресу (фізичну сторінку). І нарешті, в тих випадках, коли сторінка, до якої звертається процес, не знаходиться у фізичній пам'яті, потрібно організувати її підкачку з диска. Для контролю наявності сторінки в пам'яті вводиться спеціальний біт присутності, що входить до складу атрибутів сторінки в таблиці сторінок.

Таким чином, в наявності всіх компонентів процесу в основній пам'яті необхідності немає. Важливим наслідком такої організації є те, що розмір пам'яті, займаної процесом, може бути більше, ніж розмір оперативної пам'яті. Принцип локальності забезпечує цій схемі потрібну ефективність.

Можливість виконання програми, що знаходиться в пам'яті лише частково, має ряд цілком очевидних переваг.

- Програма не обмежена об'ємом фізичної пам'яті. Спрощується розробка програм, оскільки можна задіювати великі віртуальні простори, не піклуючись про розмір використовуваної пам'яті.
- Оскільки з'являється можливість часткового приміщення програми (процесу) в пам'ять і гнучкого перерозподілу пам'яті між програмами, можна розмістити в пам'яті більше програм, що збільшує завантаження процесора і пропускну спроможність системи.
- Об'єм введення-виводу для вивантаження частини програми на диск може бути менше, ніж у варіанті класичного свопінгу, у результаті кожна програма працюватиме швидше.

Таким чином, можливість забезпечення (за підтримки операційної системи) для програми «видимості» практично необмеженої (характерний розмір для 32-розрядної архітектури 232 = 4 Гбайт) призначеної для користувача пам'яті (логічний адресний простір), що адресується, за наявності основної пам'яті істотно менших розмірів (фізичний адресний простір) – дуже важливий аспект.

Але введення віртуальної пам'яті дозволяє вирішувати іншу, не менш важливе завдання – забезпечення контролю доступу до окремих сегментів пам'яті і, зокрема, захист призначених для користувача програм один від одного і захист ОС від призначених для користувача програм. Кожен процес працює зі своїми віртуальними адресами, трансляцію яких у фізичних виконує апаратура комп'ютера. Таким чином, призначений для користувача процес позбавлений можливості безпосередньо звернутися до сторінок основної пам'яті, зайнятих інформацією, що відноситься до інших процесів.

Наприклад, 16-розрядний комп'ютер PDP-11/70 з 64 Кбайт логічної пам'яті міг мати до 2 Мбайт оперативної пам'яті. Операційна система цього комп'ютера підтримувала віртуальну пам'ять, яка забезпечувала захист і перерозподіл основної пам'яті між призначеними для користувача процесами.

Нагадаємо, що в системах з віртуальною пам'яттю ті адреси, які генерує програма (логічні адреси), називаються віртуальними, і вони формують віртуальний адресний простір. Термін «віртуальна пам'ять» означає, що програміст має справу з пам'яттю, відмінною від реальної, розмір якої потенційно більше, ніж розмір оперативної пам'яті.

Хоча відомі і чисто програмні реалізації віртуальної пам'яті, цей напрям отримав найбільш широкий розвиток після відповідної апаратної підтримки.

Слід зазначити, що устаткування комп'ютера бере участь в трансляції адреси практично у всіх схемах управління пам'яттю. Але у разі віртуальної пам'яті це стає складнішим унаслідок розривності відображення і багатовимірності логічного адресного простору. Можливо, найбільш істотним внеском апаратури в реалізацію описуваної схеми є автоматична генерація виняткових ситуацій за відсутності в пам'яті потрібних сторінок (page fault).

Будь-яка з трьох раніше розглянутих схем управління пам'яттю – сторінковою, сегментною і сегментно-сторінковою – придатна для організації віртуальної пам'яті. Найчастіше використовується сегментно-сторінкова модель, яка є синтезом сторінкової моделі і ідеї сегментації. Причому для тієї архітектури, в якій сегменти не підтримуються апаратно, їх реалізація – завдання не-залежного компоненту менеджера пам'яті.

Сегментна організація в чистому вигляді зустрічається рідко.

Архітектурні засоби підтримки віртуальної пам'яті

Очевидно, що неможливо створити повністю машинно-незалежний компонент управління віртуальною пам'яттю. З іншого боку, є істотні частини програмного забезпечення, пов'язаного з управлінням віртуальною пам'яттю, для яких деталі апаратної реалізації абсолютно не важливі. Одним з досягнень сучасних ОС є грамотне і ефективне розділення засобів управління віртуальною пам'яттю на апаратно-незалежну і апаратно-залежну частини. Коротко розглянемо, що і яким чином входить в апаратно-залежну частину підсистеми управління віртуальною пам'яттю. Компоненти апаратно-незалежної підсистеми будуть розглянуті в наступній лекції.

У найпоширенішому випадку необхідно відобразити великий віртуальний адресний простір у фізичний адресний простір істотно меншого розміру. Призначений для користувача процес або ОС повинні мати можливість здійснити запис за віртуальною адресою, а завдання ОС – зробити так, щоб записана інформація опинилася у фізичній пам'яті (згодом при браку оперативної пам'яті вона може бути витиснена в зовнішню пам'ять). У разі віртуальної пам'яті система відображення адресних просторів окрім трансляції адрес повинна передбачати ведення таблиць, що показують, які області віртуальної пам'яті в даний момент знаходяться у фізичній пам'яті і де саме розміщуються.

Сторінкова віртуальна пам'ять

Як і у разі простої сторінкової організації, сторінкова віртуальна пам'ять і фізична пам'ять представляються такими, що складаються з наборів блоків або сторінок однакового розміру. Віртуальні адреси діляться на сторінки (page), відповідні одиниці у фізичній пам'яті утворюють сторінкові кадри (page frames), а в цілому система підтримки сторінкової віртуальної пам'яті називається пейджингом (paging). Передача інформації між пам'яттю і диском завжди здійснюється цілими сторінками.

Після розбиття менеджером пам'яті віртуального адресного простору на сторінки віртуальна адреса перетвориться у впорядковану пару (p,d), де p – номер сторінки у віртуальній пам'яті, а d – зсув в рамках сторінки p, усередині якої розміщується елемент, що адресується. Процес може виконуватися, якщо його поточна сторінка знаходиться в оперативній пам'яті. Якщо поточної сторінки в головній пам'яті немає, вона має бути переписана (підкачана) із зовнішньої пам'яті. Сторінку, що поступила, можна помістити в будь-який вільний сторінковий кадр.

Оскільки число віртуальних сторінок велике, таблиця сторінок набирає специфічного вигляду (див. розділ «Структура таблиці сторінок»), структура записів стає складнішою, серед атрибутів сторінки з'являються біти присутності, модифікації і інші керівники биті.

За відсутності сторінки в пам'яті в процесі виконання команди виникає виняткова ситуація, звана сторінкове порушення (page fault) або сторінкова відмова. Обробка сторінкового порушення полягає в тому, що виконання команди уривається, сторінка, що зажадалася, підкачується з конкретного місця вторинної пам'яті у вільний сторінковий кадр фізичної пам'яті і спроба виконання команди повторюється. За відсутності вільних сторінкових кадрів на диск вивантажується рідко використовувана сторінка. Проблеми заміщення сторінок і обробки сторінкових порушень розглядаються в наступній лекції.

Для управління фізичною пам'яттю ОС підтримує структуру таблиці кадрів. Вона має один запис на кожен фізичний кадр, що показує його стан.

У більшості сучасних комп'ютерів із сторінковою організацією в основній пам'яті зберігається лише частина таблиці сторінок, а швидкість доступу до елементів таблиці поточної віртуальної пам'яті досягається, як буде показано нижче, за рахунок використання надшвидкодіючої пам'яті, розміщеної в кеші процесора.

Сегментно-сторінкова організації віртуальної пам'яті

Як і у разі простої сегментації, в схемах віртуальної пам'яті сегмент – це лінійна послідовність адрес, що починається з 0. При організації віртуальної пам'яті розмір сегменту може бути великий, наприклад може перевищувати розмір оперативної пам'яті. Повторюючи всі раніше приведені міркування про розміщення в пам'яті великих програм, приходимо до розбиття сегментів на сторінки і необхідності підтримки своєї таблиці сторінок для кожного сегменту.

На практиці, проте, появи в системі великої кількості таблиць сторінок прагнуть уникнути, організовуючи сегменти, що не перекриваються, в одному віртуальному просторі, для опису якого вистачає однієї таблиці сторінок. Таким чином, одна таблиця сторінок відводиться для всього процесу. Наприклад, в популярних ОС Linux і Windows 2000 всіх сегментів процесу, а також область пам'яті ядра обмежено віртуальним адресним простором об'ємом 4 Гбайт. При цьому ядро ОС розташовується по фіксованих віртуальних адресах незалежно від виконуваного процесу.

Структура таблиці сторінок

Організація таблиці сторінок – один з ключових елементів відображення адрес в сторінковій і сегментно-сторінковій схемах. Розглянемо структуру таблиці сторінок для випадку сторінкової організації детальніше.

Отже, віртуальна адреса складається з віртуального номера сторінки і зсуву. Номер запису в таблиці сторінок відповідає номеру віртуальної сторінки. Розмір запису коливається від системи до системи, але найчастіше він складає 32 бита. З цього запису в таблиці сторінок знаходиться номер кадру для даної віртуальної сторінки, потім додається зсув і формується

фізична адреса. Окрім цього запис в таблиці сторінок містить інформацію про атрибути сторінки. Це біти присутності і захисту (наприклад, 0 – read/write, 1 – read only...). Також можуть бути вказані: біт модифікації, який встановлюється, якщо вміст сторінки модифікований, і дозволяє контролювати необхідність перезапису сторінки на диск; біт посилання, який допомагає виділити сторінки, які мало використовуються; біт, що вирішує кешування, і інші керівники бітів. Відмітимо, що адреси сторінок на диску не є частиною таблиці сторінок.

Основну проблему для ефективної реалізації таблиці сторінок створюють великі розміри віртуальних адресних просторів сучасних комп'ютерів, які зазвичай визначаються розрядністю архітектури процесора. Найпоширенішими на сьогодні є 32-розрядні процесори, що дозволяють створювати віртуальні адресні простори розміром 4 Гбайт (для 64-розрядних комп'ютерів ця величина дорівнює 264 байт). Крім того, існує проблема швидкості відображення, яка вирішується за рахунок використання так званої асоціативної пам'яті (див. наступний розділ).

Підрахуємо зразковий розмір таблиці сторінок. У 32-бітовому адресному просторі при розмірі сторінки 4 Кбайт (Intel) отримуємо 232/212=220, тобто приблизно мільйон сторінок, а в 64-бітовому і того більш. Таким чином, таблиця повинна мати приблизно мільйон рядків (entry), причому запис в рядку складається з декількох байтів. Відмітимо, що кожен процес потребує своєї таблиці сторінок (а у разі сегментно-сторінкової схеми бажано мати по одній таблиці сторінок на кожен сегмент).

Зрозуміло, що кількість пам'яті, що відводиться таблицям сторінок, не може бути таке велике. Для того, щоб уникнути розміщення в пам'яті величезної таблиці, її розбивають на ряд фрагментів. У оперативній пам'яті зберігають лише деякі, необхідні для конкретного моменту виконання фрагменти таблиці сторінок. Через властивість локальності число таких фрагментів відносно невелике. Виконати розбиття таблиці сторінок на частини можна по-різному. Найбільш поширений спосіб розбиття – організація так званої багаторівневої таблиці сторінок. Для прикладу розглянемо дворівневу таблицю з розміром сторінок 4 Кбайт, реалізовану в 32-розрядній архітектурі Intel.

Таблиця, що складається з 220 рядків, розбивається на 210 таблиць другого рівня по 210 рядків. Ці таблиці другого рівня об'єднані в загальну структуру за допомогою однієї таблиці першого рівня, що складається з 210 рядків. 32-розрядна адреса ділиться на 10-розрядне поле p1, 10-розрядне поле p2 і 12-розрядний зсув d. Поле p1 указує на потрібний рядок в таблиці першого рівня, поле p2 – другого, а поле d локалізує потрібний байт усередині вказаного сторінкового кадру (див. мал. 11.1).

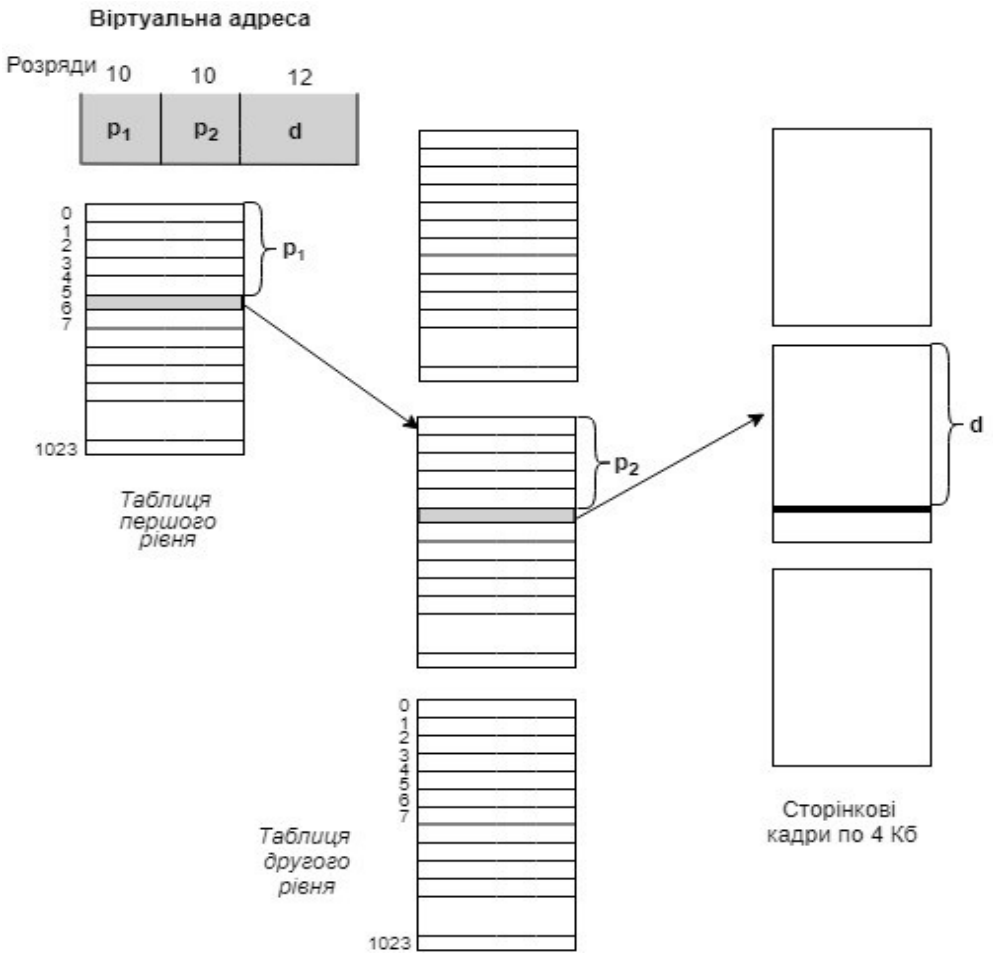


Рис. 11.1. Приклад дворівневої таблиці сторінок

За допомогою всього лише однієї таблиці другого рівня можна охопити 4 Мбайт (4 Кбайт x 1024) оперативної пам'яті. Таким чином, для розміщення процесу з великим об'ємом займаної пам'яті досить мати в оперативній пам'яті одну таблицю першого рівня і декілька таблиць другого рівня. Очевидно, що сумарна кількість рядків в цих таблицях багато менше 220. Такий підхід природним чином узагальнюється на три і більш за рівні таблиці.

Наявність декількох рівнів знижує продуктивність менеджера пам'яті. Не дивлячись на те, що розміри таблиць на кожному рівні підібрані так, щоб таблиця поміщалася цілком усередині однієї сторінки, звернення до кожного рівня – це окреме звернення до пам'яті. Таким чином, трансляція адреси може зажадати декількох звернень до пам'яті.

Кількість рівнів в таблиці сторінок залежить від конкретних особливостей архітектури. Можна привести приклади реалізації однорівневого (DEC PDP-11), дворівневого (Intel, DEC VAX), трирівневого (Sun SPARC, DEC Alpha) пейджинга, а також пейджинга із заданою кількістю рівнів (Motorola). Функціонування RISC-процессора MIPS R2000 здійснюється взагалі без таблиці сторінок. Тут пошук потрібної сторінки, якщо ця сторінка відсутня в асоціативній пам'яті, повинна узяти на себе ОС (так званий zero level paging).

Асоціативна пам'ять

Пошук номера кадру, відповідного потрібній сторінці, в багаторівневій таблиці сторінок вимагає декількох звернень до основної пам'яті, тому займає багато часу. В деяких випадках така затримка недопустима. Проблема прискорення пошуку вирішується на рівні архітектури комп'ютера.

Відповідно до властивості локальності більшість програм протягом деякого проміжку часу звертаються до невеликої кількості сторінок, тому активно використовується тільки невелика частина таблиці сторінок.

Вирішення проблеми прискорення – забезпечити комп'ютер апаратним пристроєм для відображення віртуальних сторінок у фізичних без звернення до таблиці сторінок, тобто мати невелику, швидку кеш-пам'ять, що зберігає необхідну на даний момент частину таблиці сторінок. Цей пристрій називається асоціативною пам'яттю, іноді також використовують термін буфер пошуку трансляції (translation lookaside buffer – TLB).

Один запис таблиці в асоціативній пам'яті (один вхід) містить інформацію про одну віртуальну сторінку: її атрибути і кадр, в якому вона знаходиться. Ці поля в точності відповідають полям в таблиці сторінок.

Оскільки асоціативна пам'ять містить тільки деякі із записів таблиці сторінок, кожен запис в TLB повинен включати поле з номером віртуальної сторінки. Пам'ять називається асоціативною, тому що в ній відбувається одночасне порівняння номера віртуальної сторінки, що відображається, з відповідним полем у всіх рядках цієї невеликої таблиці. Тому даний вид пам'яті досить дорога вартість. У рядку, поле віртуальної сторінки якої збіглося з шуканим значенням, знаходиться номер сторінкового кадру. Звичайне число записів в TLB від 8 до 4096. Зростання кількості записів в асоціативній пам'яті повинне здійснюватися з урахуванням таких чинників, як розмір кешу основної пам'яті і кількості звернень до пам'яті при виконанні однієї команди.

Розглянемо функціонування менеджера пам'яті за наявності асоціативної пам'яті.

Спочатку інформація про відображення віртуальної сторінки у фізичну відшукується в асоціативній пам'яті. Якщо потрібний запис знайдений – все нормально, за винятком випадків порушення привілеїв, коли запит на звернення до пам'яті відхиляється.

Якщо потрібний запис в асоціативній пам'яті відсутній, відображення здійснюється через таблицю сторінок. Відбувається заміна одного із записів в асоціативній пам'яті знайденим записом з таблиці сторінок. Тут ми стикаємося з традиційною для будь-якого кеша проблемою заміщення (а саме який із записів в кеші необхідно змінити). Конструкція асоціативної пам'яті повинна організовувати записи так, щоб можна було ухвалити рішення про те, який із старих записів має бути видалена при внесенні нових.

Число вдалих пошуків номера сторінки в асоціативній пам'яті по відношенню до загального числа пошуків називається hit (збіг) ratio (пропорція, відношення). Іноді також використовується термін «відсоток попадань в кеш». Таким чином, hit ratio – частина посилань, яка може бути зроблена з використанням асоціативної пам'яті. Звернення до одних і тих же сторінок підвищує hit ratio. Чим більше hit ratio, тим менше середній час доступу до даних, що знаходяться в оперативній пам'яті.

Припустимо, наприклад, що для визначення адреси у разі кеш-промаху через таблицю сторінок необхідно 100 нс, а для визначення адреси у разі кеш-попадання через асоціативну пам'ять – 20 нс. З 90% hit ratio середній час визначення адреси – 0,9x20+0,1x100 = 28 нс.

Цілком прийнятна продуктивність сучасних ОС доводить ефективність використання асоціативної пам'яті. Високе значення вірогідності знаходження даних в асоціативній пам'яті пов'язане з наявністю у даних об'єктивних властивостей: просторовій і тимчасовій локальності.

Необхідно звернути увагу на наступний факт. При перемиканні контексту процесів потрібно добитися того, щоб новий процес «не бачив» в асоціативній пам'яті інформацію, що відноситься

до попереднього процесу, наприклад очищати її. Таким чином, використання асоціативної пам'яті збільшує час перемикання контексту.

Розглянута дворівнева (асоціативна пам'ять + таблиця сторінок) схема перетворення адреси є яскравим прикладом ієрархії пам'яті, заснованої на використанні принципу локальності, про що мовилося у введенні до попередньої лекції.

Інвертована таблиця сторінок

Не дивлячись на багаторівневу організацію, зберіганням декількох таблиць сторінок великого розміру як і раніше є проблему. Її значення особливе актуально для 64-розрядної архітектури, де число віртуальних сторінок дуже велике. Варіантом рішення є застосування інвертованої таблиці сторінок (inverted page table). Цей підхід застосовується на машинах POWERPC, деяких робочих станціях Hewlett-Packard, IBM RT, IBM AS/400 і ряду інших.

У цій таблиці міститься по одному запису на кожен сторінковий кадр фізичної пам'яті. Істотно, що достатньо однієї таблиці для всіх процесів. Таким чином, для зберігання функції відображення потрібна фіксована частина основної пам'яті, незалежно від розрядності архітектури, розміру і кількості процесів. Наприклад, для комп'ютера Pentium з 256 Мбайт оперативної пам'яті потрібна таблиця розміром 64 Кбайт рядків.

Не дивлячись на економію оперативної пам'яті, застосування інвертованої таблиці має істотний мінус – записи в ній (як і в асоціативній пам'яті) не відсортовані за збільшенням номерів віртуальних сторінок, що ускладнює трансляцію адреси. Один із способів вирішення даної проблеми – використання хеш-таблиці віртуальних адрес. При цьому частина віртуальної адреси, що є номером сторінки, відображається в хеш-кодування-таблицю з використанням функції хешування. Кожній сторінці фізичної пам'яті тут відповідає один запис в хеш-таблиці і інвертованій таблиці сторінок. Віртуальні адреси, що мають одне значення хеш-функції, зчіплюються один з одним. Зазвичай довжина ланцюжка не перевищує двох записів.

Розмір сторінки

Розробники ОС для існуючих машин рідко мають можливість впливати на розмір сторінки. Проте для новостворюваних комп'ютерів вирішення щодо оптимального розміру сторінки є актуальним. Як і слід було чекати, немає одного якнайкращого розміру. Швидше є набір чинників, що впливають на розмір. Зазвичай розмір сторінки – це ступінь двійки від 29 до 214 байт.

Чим більше розмір сторінки, тим менше буде розмір структур даних, обслуговуючих перетворення адрес, але тим більше будуть втрати, пов'язані з тим, що пам'ять можна виділяти тільки посторінковий.

Як слід вибирати розмір сторінки? По-перше, потрібно враховувати розмір таблиці сторінок, тут бажаний великий розмір сторінки (сторінок менше, відповідно і таблиця сторінок менша). З іншого боку, пам'ять краще утилізувалася з маленьким розміром сторінки. В середньому половина останньої сторінки процесу пропадає. Необхідно також враховувати об'єм введення-виводу для взаємодії із зовнішньою пам'яттю і інші чинники. Проблема не має ідеального рішення. Історична тенденція полягає в збільшенні розміру сторінки.

Як правило, розмір сторінок задається апаратний, наприклад в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в іншій архітектурі, таких як Motorola 68030, розмір сторінок може бути заданий програмно. Враховуючи всі обставини, у ряді архітектури виникають множинні розміри сторінок, наприклад в Pentium розмір сторінки коливається від 4 Кбайт до 8 Кбайт. Проте більшість комерційних ОС зважаючи на складність переходу на множинний розмір сторінок підтримують тільки один розмір сторінок.

Висновок

У цій лекції розглянуті апаратні особливості підтримки віртуальної пам'яті. Розбиття адресного простору процесу на частини і динамічна трансляція адреси дозволили виконувати процес навіть у відсутність деяких його компонентів в оперативній пам'яті. Підкачка бракуючих компонентів з диска здійснюється операційною системою в той момент, коли в них виникає необхідність. Наслідком такої стратегії є можливість виконання великих програм, розмір яких може перевищувати розмір оперативної пам'яті. Щоб забезпечити даній схемі потрібну продуктивність, відображення адрес здійснюється апаратних за допомогою багаторівневої таблиці сторінок і асоціативної пам'яті.

Більшість ОС використовують сегментно-сторінкову віртуальну пам'ять. Для забезпечення потрібної продуктивності менеджер пам'яті ОС прагне підтримувати в оперативній пам'яті актуальну інформацію, намагаючись вгадати, до яких логічних адрес надійде звернення в недалекому майбутньому. Вирішальну роль тут грає вдалий вибір стратегії заміщення, реалізованої в алгоритмі виштовхування сторінок.

Виняткові ситуації при роботі з пам'яттю

З матеріалу попередньої лекції виходить, що відображення віртуальної адреси у фізичну здійснюється за допомогою таблиці сторінок. Для кожної віртуальної сторінки запис в таблиці сторінок містить номер відповідного сторінкового кадру в оперативній пам'яті, а також атрибути сторінки для контролю звернень до пам'яті.

Що ж відбувається, коли потрібної сторінки в пам'яті немає або операція звернення до пам'яті недопустима? Операційна система має бути якось оповіщена про той, що відбувся. Зазвичай для цього використовується механізм виняткових ситуацій (exceptions). При спробі виконати подібне звернення до віртуальної сторінки виникає виняткова ситуація "сторінкове порушення" (page fault), що приводить до виклику спеціальної послідовності команд для обробки конкретного виду сторінкового порушення.

Сторінкове порушення може відбуватися в різних випадках: за відсутності сторінки в оперативній пам'яті, при спробі запису в сторінку з атрибутом "тільки читання" або при спробі читання або запису сторінки з атрибутом "тільки виконання". У будь-якому з цих випадків викликається обробник сторінкового порушення, що є частиною операційної системи. Йому зазвичай передається причина виникнення виняткової ситуації і віртуальна адреса, звернення до якого викликало порушення.

Нас цікавитиме конкретний варіант сторінкового порушення - звернення до відсутньої сторінки, оскільки саме його обробка багато в чому визначає продуктивність сторінкової системи. Коли програма звертається до віртуальної сторінки, відсутньої в основній пам'яті, операційна система повинна виділити сторінку основної пам'яті, перемістити в неї копію віртуальної сторінки із зовнішньої пам'яті і модифікувати відповідний елемент таблиці сторінок.

Підвищення продуктивності обчислювальної системи може бути досягнуте за рахунок зменшення частоти сторінкових порушень, а також за рахунок збільшення швидкості їх обробки. Час ефективного доступу до відсутньої в оперативній пам'яті сторінки складається з:

- обслуговування виняткової ситуації (page fault);
- читання (підкачки) сторінки з вторинної пам'яті (іноді, при недоліку місця в основній пам'яті, необхідно виштовхнути одну із сторінок з основної пам'яті у вторинну, тобто здійснити заміщення сторінки);
- відновлення виконання процесу, що викликав даний page fault.

Для вирішення першого і третього завдань ОС виконує до декількох сотень машинних інструкцій протягом декількох десятків мікросекунд. Час підкачки сторінки близький до декількох десяток мілісекунд. Проведені дослідження показують, що вірогідності page fault 5x10-7 виявляється достатньо, щоб понизити продуктивність сторінкової схеми управління пам'яттю на 10%. Таким чином, зменшення частоти page faults є одному з ключових завдань системи управління пам'яттю. Її рішення зазвичай пов'язане з правильним вибором алгоритму заміщення сторінок.

Стратегії управління сторінковою пам'яттю

Програмне забезпечення підсистеми управління пам'яттю пов'язане з реалізацією наступних стратегій:

Стратегія вибірки (fetch policy) - в який момент слід переписати сторінку з вторинної пам'яті в первинну. Існує два основні варіанти вибірки - за запитом і з попередженням. Алгоритм вибірки за запитом вступає в дію в той момент, коли процес звертається до відсутньої сторінки, вміст якої знаходиться на диску. Його реалізація полягає в завантаженні сторінки з диска у вільну фізичну сторінку і корекції відповідного запису таблиці сторінок.

Алгоритм вибірки з попередженням здійснює випереджаюче читання, тобто окрім сторінки, що викликала виняткову ситуацію, в пам'ять також завантажується декілька сторінок, що оточують її (зазвичай сусідні сторінки розташовуються в зовнішній пам'яті послідовно і можуть бути лічені за одне звернення до диска). Такий алгоритм покликаний зменшити накладні витрати, пов'язані з великою кількістю виняткових ситуацій, що виникають при роботі із значними об'ємами даних або коди; крім того, оптимізується робота з диском.

Стратегія розміщення (placement policy) - в яку ділянку первинної пам'яті помістити сторінку, що поступає. У системах із сторінковою організацією все просто - в будь-який вільний сторінковий кадр. У разі систем з сегментною організацією необхідна стратегія, аналогічна стратегії з динамічним розподілом.

Стратегія заміщення (replacement policy) - яку сторінку потрібно виштовхнути в зовнішню пам'ять, щоб звільнити місце в оперативній пам'яті. Розумна стратегія заміщення, реалізована у відповідному алгоритмі заміщення сторінок, дозволяє зберігати в пам'яті найнеобхіднішу інформацію і тим самим понизити частоту сторінкових порушень. Заміщення повинне відбуватися з урахуванням виділеного кожному процесу кількості кадрів. Крім того, потрібно вирішити, чи повинна сторінка, що заміщається, належати процесу, який ініціював заміщення, або вона має бути вибрана серед всіх кадрів основної пам'яті.

Алгоритми заміщення сторінок

Отже, найбільш відповідальною дією менеджера пам'яті є виділення кадру оперативній пам'яті для розміщення в ній віртуальної сторінки, що знаходиться в зовнішній пам'яті. Нагадаємо, що ми розглядаємо ситуацію, коли розмір віртуальної пам'яті для кожного процесу може істотно перевершувати розмір основної пам'яті. Це означає, що при виділенні сторінки основній пам'яті з великою вірогідністю не вдасться знайти вільний сторінковий кадр. В цьому випадку операційна система відповідно до закладених в неї критеріїв повинна:

- знайти деяку зайняту сторінку основної пам'яті;
- перемістити у разі потреби її вміст в зовнішню пам'ять;
- переписати в цей сторінковий кадр вміст потрібної віртуальної сторінки із зовнішньої пам'яті;
- належним чином модифікувати необхідний елемент відповідної таблиці сторінок;
- продовжити виконання процесу, якому ця віртуальна сторінка знадобилася.

Відмітимо, що при заміщенні доводиться двічі передавати сторінку між основною і вторинною пам'яттю. Процес заміщення може бути оптимізований за рахунок використання біта модифікації (один з атрибутів сторінки в таблиці сторінок). Біт модифікації встановлюється комп'ютером, якщо хоч би один байт був записаний на сторінку. При виборі кандидата на заміщення перевіряється біт модифікації. Якщо біт не встановлений, немає необхідності переписувати дану сторінку на диск, її копія на диску вже є. Подібний метод також

застосовується до read-only-сторінкам, вони ніколи не модифікуються. Ця схема зменшує час обробки page fault.

Існує велика кількість різноманітних алгоритмів заміщення сторінок. Всі вони діляться на локальних і глобальних. Локальні алгоритми, на відміну від глобальних, розподіляють фіксоване або таке, що динамічно налаштовує число сторінок для кожного процесу. Коли процес витратить всі призначені йому сторінки, система видалятиме з фізичної пам'яті одну з його сторінок, а не із сторінок інших процесів. Глобальний алгоритм заміщення у разі виникнення виняткової ситуації задовільниться звільненням будь-якої фізичної сторінки, незалежно від того, якому процесу вона належала.

Глобальні алгоритми мають ряд недоліків. По-перше, вони роблять одні процеси чутливими до поведінки інших процесів. Наприклад, якщо один процес в системі одночасно використовує велику кількість сторінок пам'яті, то решта всіх застосувань в результаті відчуватиме сильне уповільнення через нестачу кадрів пам'яті для своєї роботи. По-друге, некоректно працюючe застосування може підірвати роботу всієї системи (якщо, звичайно, в системі не передбачено обмеження на розмір пам'яті, що виділяється процесу), намагаючись захопити більше пам'яті. Тому в багатозадачній системі іноді доводиться використовувати складніші локальні алгоритми. Застосування локальних алгоритмів вимагає зберігання в операційній системі списку фізичних кадрів, виділених кожному процесу. Цей список сторінок іноді називають резидентним безліччю процесу. У одному з наступних розділів розглянутий варіант алгоритму підкачки, заснований на приведенні резидентної множини у відповідність так званому робочому набору процесу.

Ефективність алгоритму зазвичай оцінюється на конкретній послідовності посилань до пам'яті, для якої підраховується число тих, що виникають page faults. Ця послідовність називається рядком звернень (reference string). Ми можемо генерувати рядок звернень штучним чином за допомогою датчика випадкових чисел або трасуючи конкретну систему. Останній метод дають дуже багато посилання, для зменшення числа яких можна зробити дві речі:

- для конкретного розміру сторінок можна запам'ятовувати тільки їх номери, а не адреси, на які йде посилання;
- декілька підряд посилань, що йдуть, на одну сторінку можна фіксувати один раз.

Як вже мовилося, більшість процесорів мають прості апаратні засоби, що дозволяють збирати деяку статистику звернень до пам'яті. Ці засоби зазвичай включають два спеціальні прапори на кожен елемент таблиці сторінок. Прапор посилання (reference) автоматично встановлюється, коли відбувається будь-яке звернення до цієї сторінки, а вже розглянутий вище прапор зміни (modify) встановлюється, якщо проводиться запис в цю сторінку. Операційна система періодично перевіряє встановлення таких прапорів, для того, щоб виділити активно використовувані сторінки, після чого значення цих прапорів скидаються.

Розглянемо ряд алгоритмів заміщення сторінок.

Алгоритм FIFO. Виштовхування першої сторінки, що прийшла

Простий алгоритм. Кожній сторінці привласнюється тимчасова мітка. Реалізується це просто створенням черги сторінок, в кінець якої сторінки потрапляють, коли завантажуються у фізичну пам'ять, а з початку беруться, коли потрібно звільнити пам'ять. Для заміщення вибирається стара сторінка. На жаль, ця стратегія з достатньою вірогідністю приводитиме до заміщення активно використовуваних сторінок, наприклад сторінок коди текстового процесора при редагуванні файлу. Відмітимо, що при заміщенні активних сторінок все працює коректно, але page fault відбувається негайно.

Аномалія Біледі (Belady)

На перший погляд здається очевидним, що чим більше в пам'яті сторінкових кадрів, тим рідше матимуть місце page faults. Дивно, але це не завжди так. Як встановив Біледі з колегами, певні послідовності звернень до сторінок насправді приводять до збільшення числа сторінкових порушень при збільшенні кадрів, виділених процесу. Це явище носить назва "Аномалії Біледі" або "аномалії FIFO".

Система з трьома кадрами (9 faults) виявляється продуктивнішою, ніж з чотирма кадрами (10 faults), для рядка звернень до пам'яті 012301401234 при виборі стратегії FIFO.

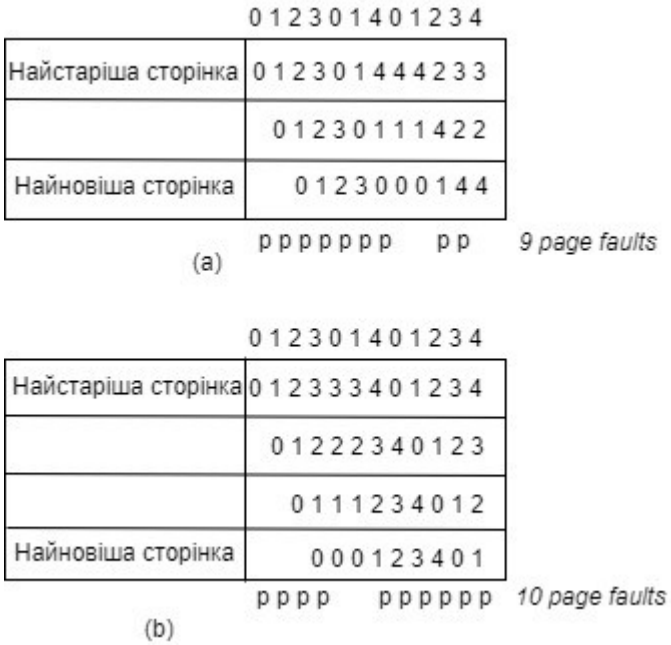


Рис. 12.1. Аномалія Біледі: (a) - FIFO з трьома сторінковими кадрами; (b) - FIFO з чотирма сторінковими кадрами

Аномалію Біледі слід рахувати швидше курйозом, чим чинником, що вимагає серйозного відношення, який ілюструє складність ОС, де інтуїтивний підхід не завжди прийнятний.

Оптимальний алгоритм (OPT)

Одним з наслідків відкриття аномалії Біледі став пошук оптимального алгоритму, який при заданому рядку звернень мав би мінімальну частоту page faults серед всіх інших алгоритмів. Такий алгоритм був знайдений. Він простий: заміщай сторінку, яка не використовуватиметься протягом найтривалішого періоду часу.

Кожна сторінка має бути помічена числом інструкцій, які будуть виконані, перш ніж на цю сторінку буде зроблено перше посилання. Виштовхуватися повинна сторінка, для якої це число найбільше.

Цей алгоритм легко описати, але реалізувати неможливо. ОС не знає, до якої сторінки буде наступне звернення. (Раніше такі проблеми виникали при плануванні процесів - алгоритм SJF).

Зате ми можемо зробити висновок, що для того, щоб алгоритм заміщення був максимально близький до ідеального алгоритму, система винна якомога точніше передбачати звернення процесів до пам'яті. Даний алгоритм застосовується для оцінки якості алгоритмів, що реалізовуються.

Виштовхування найдовшої сторінки, що не використовувалася. Алгоритм LRU

Одним з наближених до алгоритму OPT є алгоритм, випливаючий з евристичного правила, що недавнє минуле - хороший орієнтир для прогнозування найближчого майбутнього.

Ключова відмінність між FIFO і оптимальним алгоритмом полягає в тому, що один дивиться назад, а інший вперед. Якщо використовувати минуле для апроксимації майбутнього, має сенс заміщати сторінку, яка не використовувалася протягом найдовшого часу. Такий підхід називається least recently used алгоритм (LRU). Робота алгоритму проілюстрована на мал. мал. 12.2. Порівнюючи мал. 12.1 b і 12.2, можна побачити, що використання LRU алгоритму дозволяє скоротити кількість сторінкових порушень.

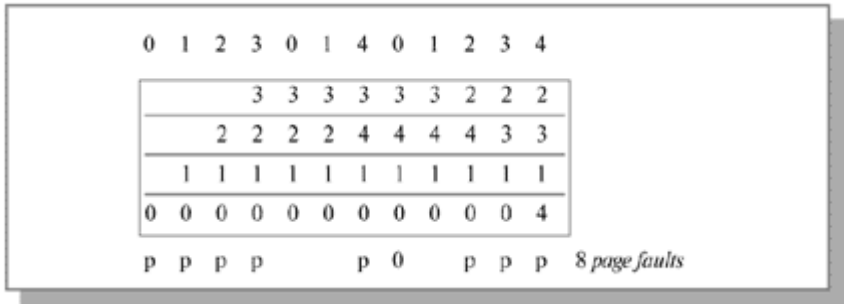


Рис. 12.2. Приклад роботи алгоритму LRU

LRU - хороший, але важкореалізовуваний алгоритм. Необхідно мати зв'язаний список всіх сторінок в пам'яті, на початку якого будуть зберігатися недавно використані сторінки. Причому цей список повинен оновлюватися при кожному зверненні до пам'яті. Багато часу потрібно і на пошук сторінок в такому списку.

У [Таненбаум, 2002] розглянутий варіант реалізації алгоритму LRU із спеціальним 64-бітовим покажчиком, який автоматично збільшується на одиницю після виконання кожної інструкції, а в таблиці сторінок є відповідне поле, в яке заноситься значення покажчика при кожному посиланні на сторінку. При виникненні page fault вивантажується сторінка з найменшим значенням цього поля.

Як оптимальний алгоритм, так і LRU не страждають від аномалії Біледі. Існує клас алгоритмів, для яких при одному і тому ж рядку звернень безліч сторінок в пам'яті для n кадрів завжди є підмножиною сторінок для n+1 кадру. Ці алгоритми не проявляють аномалії Біледі і називаються стековими (stack) алгоритмами.

Виштовхування рідко використовуваної сторінки. Алгоритм NFU

Оскільки більшість сучасних процесорів не надають відповідної апаратної підтримки для реалізації алгоритму LRU, хотілося б мати алгоритм, достатньо близький до LRU, але що не вимагає спеціальної підтримки.

Програмна реалізація алгоритму, близького до LRU, - алгоритм NFU(Not Frequently Used).

Для нього потрібні програмні лічильники, поодинці на кожну сторінку, які спочатку дорівнюють нулю. При кожному перериванні за часом (а не після кожної інструкції) операційна система сканує всі сторінки в пам'яті і у кожної сторінки зі встановленим прапором звернення збільшує на одиницю значення лічильника, а прапор звернення скидає.

Таким чином, кандидатом на звільнення виявляється сторінка з найменшим значенням лічильника, як сторінка, до якої найрідше звертались. Головний недолік алгоритму NFU полягає в тому, що він нічого не забуває. Наприклад, сторінка, до якої дуже часто звертались протягом

деякого часу, а потім звертатися перестали, все одно не буде видалена з пам'яті, тому що її лічильник містить велику величину. Наприклад, в багатопрохідних компіляторах сторінки, які активно використовувалися під час першого проходу, можуть надовго зберегти великі значення лічильника, заважаючи завантаженню корисних надалі сторінок.

На щастя, можлива невелика модифікація алгоритму, яка дозволяє йому "забувати". Достатньо, щоб при кожному перериванні за часом вміст лічильника зрушувався управо на 1 біт, а вже потім проводилося б його збільшення для сторінок зі встановленим прапором звернення.

Іншим, вже стійкішим недоліком алгоритму є тривалість процесу сканування таблиць сторінок.

Інші алгоритми

Для повноти картини можна згадати ще декілька алгоритмів.

Наприклад, алгоритм Second-Chance - модифікація алгоритму FIFO, яка дозволяє уникнути втрати часто використовуваних сторінок за допомогою аналізу прапора звернень (біта посилання) для найстарішої сторінки. Якщо прапор встановлений, то сторінка, на відміну від алгоритму FIFO, не виштовхується, а її прапор скидається, і сторінка переноситься в кінець черги. Якщо спочатку прапори звернень були встановлені для всіх сторінок (на всі сторінки посилалися), алгоритм Second-Chance перетворюється на алгоритм FIFO. Даний алгоритм використовувався в Multics і BSD Unix.

У комп'ютері Macintosh використаний алгоритм NRU (Not Recently-Used), де сторінка-"жертва" вибирається на основі аналізу бітів модифікації і посилання. Цікаві стратегії, засновані на буферизації сторінок, реалізовані в VAX/VMS і Mach.

Є також і багато інших алгоритмів заміщення. Об'єм цього курсу не дозволяє розглянути їх детально. Докладний опис різних алгоритмів заміщення можна знайти в монографіях [Дейтел, 1987], [Цикрітіс, 1977], [Таненбаум, 2002] і ін.

Управління кількістю сторінок, виділених процесу. Модель робочої множини

У стратегіях заміщення, розглянутих в попередньому розділі, простежується припущення про те, що кількість кадрів, що належать процесу, не можна збільшити. Це приводить до необхідності виштовхування сторінки. Розглянемо більш загальний підхід, що базується на концепції робочої множини, сформульованою Деннінгом [Denning, 1996].

Отже, що робити, якщо у розпорядженні процесу є недостатнє число кадрів? Чи потрібно його припинити із звільненням всіх кадрів? Що слід розуміти під достатньою кількістю кадрів?

Трешинг (Thrashing)

Хоча теоретично можливо зменшити число кадрів процесу до мінімуму, існує якесь число активно використовуваних сторінок, без якого процес часто генерує page faults. Висока частота сторінкових порушень називається трешинг (thrashing, іноді вживається російський термін "пробуксувала", див. мал. 12.3). Процес знаходиться в змозі трешингу, якщо при його роботі більше часу йде на підкачку сторінок, ніж на виконання команд. Такого роду критична ситуація виникає незалежно від конкретних алгоритмів заміщення.

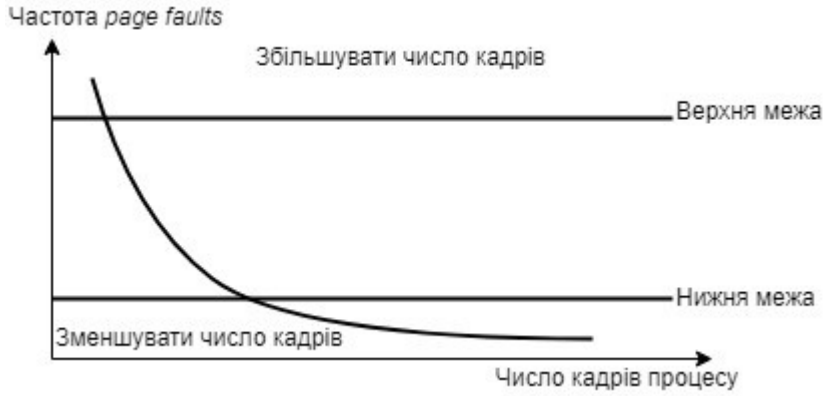


Рис. 12.3. Частота page faults залежно від кількості кадрів, виділених процесу

Часто результатом трешингу є зниження продуктивності обчислювальної системи. Один з небажаних сценаріїв розвитку подій може виглядати таким чином. При глобальному алгоритмі заміщення процес, якому не вистачає кадрів, починає відбирати кадри у інших процесів, які у свою чергу починають займатися тим же. В результаті всі процеси потрапляють в чергу запитів до пристрою вторинної пам'яті (знаходяться в стані очікування), а черга процесів в стані готовності пустує. Завантаження процесора знижується. Операційна система реагує на це збільшенням ступеня мультипрограмування, що приводить до ще більшого трешингу і подальшого зниження завантаження процесора. Таким чином, пропускна спроможність системи падає через трешинг.

Ефект трешингу, що виникає при використанні глобальних алгоритмів, може бути обмежений за рахунок застосування локальних алгоритмів заміщення. При локальних алгоритмах заміщення якщо навіть один з процесів потрапив в трешинг, це не позначається на інших процесах. Проте він багато часу проводить в черзі до пристрою вивантаження, утрудняючи підкачку сторінок решти процесів.

Критична ситуація типу трешингу виникає незалежно від конкретних алгоритмів заміщення. Єдиним алгоритмом, що теоретично гарантує відсутність трешингу, є розглянутий вище оптимальний алгоритм, що не реалізовується на практиці.

Отже, трешинг - це висока частота сторінкових порушень. Необхідно її контролювати. Коли вона висока, процес потребує кадрів. Можна, встановлюючи бажану частоту page faults, регулювати розмір процесу, додаючи або віднімаючи у нього кадри. Може виявитися доцільним вивантажити процес цілком. Кадри, що звільнилися, виділяються іншим процесам з високою частотою page faults.

Для запобігання трешингу потрібно виділяти процесу стільки кадрів, скільки йому потрібно. Але як дізнатися, скільки йому потрібно? Необхідно спробувати з'ясувати, як багато кадрів процес реально використовує. Для вирішення цього завдання Деннінг використовував модель робочої множини, яка заснована на застосуванні принципу локальності.

Модель робочої множини

Розглянемо поведінку реальних процесів.

Процеси починають працювати, не маючи в пам'яті необхідних сторінок. В результаті при виконанні першої ж машинної інструкції виникає page fault, що вимагає підкачки порції коди. Наступний page fault відбувається при локалізації глобальних змінних і ще один - при виділенні пам'яті для стека. Після того, як процес зібрав велику частину необхідних йому сторінок, page faults виникають рідко.

Таким чином, існує набір сторінок (P1, P2 ... Pn), що активно використовуються разом, який дозволяє процесу у момент часу t протягом деякого періоду T продуктивно працювати, уникаючи великої кількості page faults. Цей набір сторінок називається робочим набором W(t,T) (working set) процесу. Число сторінок в робочій множині визначається параметром T, є неубутною функцією T і відносно невеликим. Іноді T називають розміром вікна робочої множини, через яку ведеться спостереження за процесом (див. мал. 12.4).

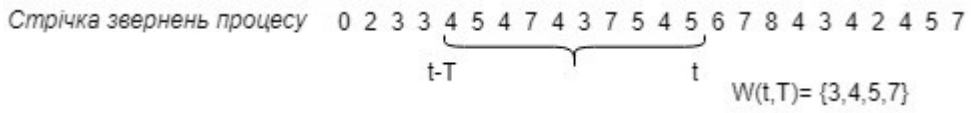


Рис. 12.4. Приклад робочої множини процесу

Легко написати тестову програму, яка систематично працює з великим діапазоном адрес, але, на щастя, більшість реальних процесів не ведуть себе так само, а проявляють властивість локальності. Протягом будь-якої фази обчислень процес працює з невеликою кількістюW сторінок.

Коли процес виконується, він рухається від однієї робочої множини до іншого. Програма зазвичай складається з декількох робочих множин, які можуть перекриватися. Наприклад, коли викликана процедура, визначає нову робочу множину, що складається із сторінок, що містять інструкції процедури, її локальні і глобальні змінні. Після її завершення процес покидає ця робоча множина, але може повернутися до нього при новому виклику процедури. Таким чином, робоча множина визначається кодом і даними програми. Якщо процесу виділяти менше кадрів, чим йому вимагається для підтримки робочої множини, він знаходитиметься в змозі трешинга.

Принцип локальності посилань перешкоджає частим змінам робочих наборів процесів. Формально це можна виразити таким чином. Якщо в період часу програма зверталася до сторінок W(t,T), то при належному виборі T з великою ймовірністю ця програма звертатиметься до тих же сторінок в період часу (t, t+T). Іншими словами, принцип локальності стверджує, що якщо не дуже далеко заглядати в майбутнє, то можна достатньо точно його прогнозувати виходячи з минулого. Зрозуміло, що з часом робочий набір процесу може змінюватися (як по складу сторінок, так і по їх числу).

Найбільш важлива властивість робочої множини - його розмір. ОС повинна виділити кожному процесу достатнє число кадрів, щоб помістилася його робоча множина. Якщо кадри ще залишилися, то може бути ініційований інший процес. Якщо робоча безліч процесів не поміщається в пам'ять і починається трешинг, то один з процесів можна вивантажити на диск.

Вирішення про розміщення процесів в пам'яті повинне, отже, базуватися на розмірі його робочої множини. Для процесів, що вперше ініціюються, це рішення може бути ухвалене евристично. Під час роботи процесу система повинна уміти визначати: розширює процес своя робоча множина або переміщається на нову робочу множину. Якщо до складу атрибутів сторінки включити час останнього використання ti (для сторінки з номером i), то приналежність i-й сторінки до робочого набору, визначуваного параметром T у момент часу t виражатиметься нерівністю: t-T < ti < t. Алгоритм виштовхування сторінок WSClock, що використовує інформацію про робочий набір процесу, описаний в [Таненбаум, 2002].

Інший спосіб реалізації даного підходу може бути заснований на відстежуванні кількості сторінкових порушень, що викликаються процесом. Якщо процес часто генерує page faults і пам'ять не дуже заповнена, то система може збільшити число виділених йому кадрів. Якщо ж процес не викликає виняткових ситуацій протягом деякого часу і рівень генерації нижче якогось

порогу, то число кадрів процесу може бути урізане. Цей спосіб регулює лише розмір безлічі сторінок, що належать процесу, і має бути доповнений якою-небудь стратегією заміщення сторінок. Не дивлячись на те що система при цьому може пробуксовувати в моменти переходу від однієї робочої множини до іншого, пропоноване рішення в змозі забезпечити якнайкращу продуктивність для кожного процесу, не вимагаючи ніякого додаткового налаштування системи.

Сторінкові демони

Підсистема віртуальної пам'яті працює продуктивно за наявності резерву вільних сторінкових кадрів. Алгоритми, що забезпечують підтримку системи в стані відсутності трешингу, реалізовані у складі фонових процесів (їх часто називають демонами або сервісами), які періодично "прокидаються" і інспектують стан пам'яті. Якщо вільні кадрів виявляється мало, вони можуть змінити стратегію заміщення. Їх завдання - підтримувати систему в стані якнайкращої продуктивності.

Прикладом такого роду процесу може бути фоновий процес - складальник сторінок, що реалізовує полегшений варіант алгоритму відкачування, заснований на використанні робочого набору і вживаний в багатьох клонах ОС Unix (див., наприклад[Bach, 1986]). Даний демон проводить відкачування сторінок, що не входять в робочі набори процесів. Він починає активно працювати, коли кількість сторінок в списку вільних сторінок досягає встановленого нижнього порогу, і намагається виштовхувати сторінки відповідно до власної стратегії.

Але якщо виникає вимога сторінки в умовах, коли список вільних сторінок порожній, то починає працювати механізм свопінгу, оскільки просте відняття сторінки у будь-якого процесу (включаючи той, який зажадав би сторінку) потенційно вело б до ситуації thrashing, і руйнувало б робочий набір деякого процесу. Будь-який процес, що зажадав сторінку не зі свого поточного робочого набору, стає в чергу на вивантаження з розрахунку на те, що після завершення вивантаження хоч би одного з процесів вільної пам'яті вже може бути досить.

У ОС Windows 2000 аналогічну роль грає менеджер балансного набору (Working set manager), який викликається раз в секунду або тоді, коли розмір вільної пам'яті опускається нижче певної межі, і відповідає за сумарну політику управління пам'яттю і підтримку робочих множин.

Програмна підтримка сегментної моделі пам'яті процесу

Реалізація функцій операційної системи, пов'язаних з підтримкою пам'яті, - ведення таблиць сторінок, трансляція адреси, обробка сторінкових помилок, управління асоціативною пам'яттю і ін. - тісно пов'язана із структурами даних, що забезпечують зручне представлення адресного простору процесу. Формат цих структур сильно залежить від апаратури і особливостей конкретної ОС.

Найчастіше віртуальна пам'ять процесу ОС розбивається на сегменти п'яти типів: коди програми, даних, стека, що розділяється і сегмент файлів, що відображаються в пам'ять (див. мал. 12.5).

Сегмент програмної коди містить тільки команди. Сегмент програмної коди не модифікується в ході виконання процесу, зазвичай сторінки даного сегменту мають атрибут read-only. Наслідком цього є можливість використання одного екземпляра коди для різних процесів.

Сегмент даних, що містить змінні програми і сегмент стеку, що містить автоматичні змінні, можуть динамічно міняти свій розмір (зазвичай дані у бік збільшення адрес, а стік - у бік зменшення) і вміст, мають бути доступні по читанню і запису і є приватними сегментами процесу.

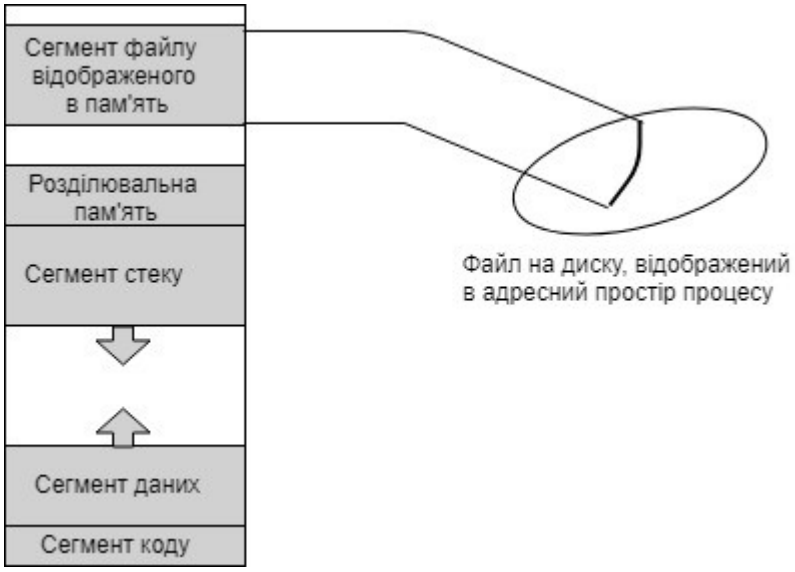


Рис. 12.5. Образ процесу в пам'яті

З метою усуспільнення пам'яті між декількома процесами створюються сегменти, що розділяються, допускають доступ по читанню і запису. Варіантом сегменту, що розділяється, може бути сегмент файлу, що відображається в пам'ять. Специфіка таких сегментів полягає в тому, що з них відкачування здійснюється не в системну область вивантаження, а безпосередньо у файл, що відображається. Реалізація сегментів, що розділяються, заснована на тому, що логічні сторінки різних процесів зв'язуються з одними і тими ж сторінковими кадрами.

Сегментами є безперервні області (у Linux вони так і називаються - області) у віртуальному адресному просторі процесу, вирівняні по межах сторінок. Кожна область складається з набору сторінок з одним і тим же режимом захисту. Між областями у віртуальному просторі можуть бути вільні ділянки. Природно, що подібні об'єкти описані відповідними структурами (див., наприклад, структури mm_struct і vm_area_struct в Linux).

Частина роботи по організації сегментів може відбуватися за участю програміста. Особливо це помітно при низькорівневому програмуванні. Зокрема, окремі області пам'яті можуть бути пойменовані і використовуватися для обміну даними між процесами. Два процеси можуть спілкуватися через область пам'яті, що розділяється, за умови, що їм відоме її ім'я (пароль). Зазвичай це робиться за допомогою спеціальних викликів (наприклад, map і unmap), що входять до складу інтерфейсу віртуальної пам'яті.

Завантаження виконуваного файлу (системний виклик exec) здійснюється зазвичай через відображення (mapping) його частин (коди, даних) у відповідні сегменти адресного простору процесу. Наприклад, сегмент коди є сегментом файлу, що відображається в пам'ять, містить виконувану програму. При спробі виконати першу ж інструкцію система виявляє, що потрібної частини коди в пам'яті немає, генерує page fault і підкачує цю частину коди з диска. Далі процедура повторюється до тих пір, поки вся програма не опиниться в оперативній пам'яті.

Як вже мовилося, розмір сегменту даних динамічно міняється. Розглянемо, як організована підтримка сегментів даних в Unix. Користувач, запрошуючи (бібліотечні виклики malloc, new) або звільняючи (free, delete) пам'ять для динамічних даних, фактично змінює межу виділеною процесу пам'яті через системний виклик brk (від слова break), який модифікує значення змінної brk із структури даних процесу. В результаті відбувається виділення фізичній пам'яті, межа brk зміщується у бік збільшення віртуальних адрес, а відповідні рядки таблиць сторінок отримують осмислені значення. За допомогою того ж виклику brk користувач може зменшити розмір сегменту даних. На практиці звільнена користувачем віртуальна пам'ять (бібліотечні виклики free, delete) системі не повертається. На це є дві причини. По-перше, для зменшення розмірів

сегменту даних необхідно організувати його ущільнення або "збірку сміття". А по-друге, незайняті усередині сегменту даних області природним чином виштовхнуть з оперативної пам'яті унаслідок того, що до них не буде звернень. Ведення списків зайнятих і вільних областей пам'яті в сегменті даних користувача здійснюється на рівні системних бібліотек.

Детальніше інформація про адресні простори процесів в Unix викладена в [Ковалів] [Bach, 1986].

Окремі аспекти функціонування менеджера пам'яті

Коректна робота менеджера пам'яті окрім принципових питань, пов'язаних з вибором абстрактної моделі віртуальної пам'яті і її апаратною підтримкою, забезпечується також безліччю нюансів і дрібних деталей. Як приклад такого роду компоненту розглянемо детальніше локалізацію сторінок в пам'яті, яка застосовується в тих випадках, коли підтримка сторінкової системи приводить до необхідності дозволити певним сторінкам, що зберігають буфери введення-виводу, інші важливі дані і код, бути блокованими в пам'яті.

Розглянемо випадок, коли система віртуальної пам'яті може вступити в конфлікт з підсистемою введення-виводу. Наприклад, процес може запитати введення в буфер і чекати його завершення. Управління передасться іншому процесу, який може викликати page fault і, з відмінною від нуля вірогідністю, спровокувати вивантаження тієї сторінки, куди має бути здійснений введення першим процесом. Подібні ситуації потребують додаткового контролю, особливо якщо уведення-виведення реалізоване з використанням механізму прямого доступу до пам'яті (DMA). Одне з вирішень даної проблеми - вводити дані в буфер, що не витісняється, в просторі ядра, а потім копіювати їх в призначений для користувача простір.

Друге рішення - локалізувати сторінки в пам'яті, використовуючи спеціальний біт локалізації, що входить до складу атрибутів сторінки. Локалізована сторінка заміщенню не підлягає. Біт локалізації скидається після завершення операції введення-виводу.

Інше використання біта локалізації може мати місце і при нормальному заміщенні сторінок. Розглянемо наступний ланцюг подій. Низькопріоритетний процес після тривалого очікування отримав в своє розпорядження процесор і підкачав з диска потрібну йому сторінку. Якщо він відразу після цього буде витиснений високопріоритетним процесом, останній може легко замістити знов підкачану сторінку низькопріоритетного, оскільки на неї не було посилань. Має сенс знов завантажені сторінки позначати бітом локалізації до першого посилання, інакше низькопріоритетний процес так і не почне працювати.

Використання біта локалізації може бути небезпечним, якщо забути його відключити. Якщо така ситуація має місце, сторінка стає невживаною. SUNOS вирішує використання даного біта як підказку, яку можна ігнорувати, коли пул вільних кадрів стає дуже маленьким.

Іншим важливим застосуванням локалізації є її використання в системах м'якого реального часу. Розглянемо процес або нитку реального часу. Взагалі кажучи, віртуальна пам'ять - антитеза обчислень реального часу, оскільки дає непередбачувані затримки при підкачці сторінок. Тому системи реального часу майже не використовують віртуальну пам'ять. ОС Solaris підтримує як реальний час, так і розділення часу. Для вирішення проблеми page faults, Solaris дозволяє процесам повідомляти систему, які сторінки важливі для процесу, і локалізувати їх в пам'яті. В результаті можливе виконання процесу, що реалізовує завдання реального часу, що містить локалізовані сторінки, де тимчасові затримки сторінкової системи будуть мінімізовані.

Окрім системи локалізації сторінок, є і інші цікаві проблеми, що виникають в процесі управління пам'яттю. Так, наприклад, буває непросто здійснити повторне виконання інструкції, що викликала page fault. Представляють інтерес і алгоритми відкладеного виділення пам'яті

(копіювання при записі і ін.). Обмежений об'єм даного курсу не дозволяє розглянути їх детальніше.

Висновок

Описана система управління пам'яттю є сукупністю програмно-технічних засобів, що забезпечують продуктивне функціонування сучасних комп'ютерів. Успіх реалізації тієї частини ОС, яка відноситься до управління віртуальною пам'яттю, визначається близькістю архітектури апаратних засобів, що підтримують віртуальну пам'ять, до абстрактної моделі віртуальної пам'яті ОС. Справедливо відмітимо, що в переважній більшості сучасних комп'ютерів апаратура виконує функції, потреби моделі ОС, що істотно перевищують, так що створення апаратно-залежної частини підсистеми управління віртуальною пам'яттю ОС в більшості випадків не надмірно складною задачею.

Модуль 4. Зберігання даних у зовнішній пам’яті. Файлові системи.

Введення

Історія систем управління даними в зовнішній пам'яті починається ще з магнітних стрічок, але сучасну зовнішність вони придбали з появою магнітних дисків. До цього кожна прикладна програма сама вирішувала проблеми іменування даних і їх структуризації в зовнішній пам'яті. Це утруднювало підтримку на зовнішньому носієві декількох архівів інформації, що довготривало зберігається. Історичним кроком став перехід до використання централізованих систем управління файлами. Система управління файлами бере на себе розподіл зовнішньої пам'яті, відображення імен файлів в адреси зовнішньої пам'яті і забезпечення доступу до даних.

Файлова система – це частина операційної системи, призначення якої полягає в тому, щоб організувати ефективну роботу з даними, що зберігаються в зовнішній пам'яті, і забезпечити користувачеві зручний інтерфейс при роботі з такими даними. Організувати зберігання інформації на магнітному диску непросто. Це вимагає, наприклад, хорошого знання пристрою контроллера диска, особливостей роботи з його регістрами. Безпосередня взаємодія з диском - прерогатива компоненту системи введення-виводу ОС, званого драйвером диска. Для того, щоб позбавити користувача комп'ютера від складнощів взаємодії з апаратурою, була придумана ясна абстрактна модель файлової системи. Операції запису або читання файлу концептуально простіше, ніж низькорівневі операції роботи з пристроями.

Основна ідея використання зовнішньої пам'яті полягає в наступному. ОС ділить пам'ять на блоки фіксованого розміру, наприклад, 4096 байт. Файл, що зазвичай є неструктурованою послідовністю однобайтових записів, зберігається у вигляді послідовності блоків (не обов'язково суміжних); кожен блок зберігає ціле число записів. У деяких ОС (MS-DOS) адреси блоків, що містять дані файлу, можуть бути організовані в зв'язний список і винесені в окрему таблицю в пам'яті. У інших ОС (Unix) адреси блоків даних файлу зберігаються в окремому блоці зовнішньої пам'яті (так званому індексі або індексному вузлі). Цей прийом, званий індексацією, є найбільш поширеним для застосувань, що вимагають довільного доступу до записів файлів. Індекс файлу складається із списку елементів, кожен з яких містить номер блоку у файлі і зведення про місцеположення даного блоку. Прочитування чергового байта здійснюється з так званої поточної позиції, яка характеризується зсувом від початку файлу. Знаючи розмір блоку, легко обчислити номер блоку, що містить поточну позицію. Адресу ж потрібного блоку диска можна потім витягувати з індексу файлу. Базовою операцією, що виконується по відношенню до файлу, є читання блоку з диска і перенесення його в буфер, що знаходиться в основній пам'яті.

Файлова система дозволяє за допомогою системи довідників (каталогів, директорій) пов'язати унікальне ім'я файлу з блоками вторинної пам'яті, що містять дані файлу. Ієрархічна структура каталогів, використовувана для управління файлами, може служити іншим прикладом індексної структури. В цьому випадку каталоги або теки грають роль індексів, кожен з яких містить посилання на свої підкаталоги. З цієї точки зору вся файлова система комп'ютера є великим індексованим файлом. Окрім власне файлів і структур даних, використовуваних для управління файлами (каталоги, дескриптори файлів, різні таблиці розподілу зовнішньої пам'яті), поняття "Файлова система" включає програмні засоби, що реалізовують різні операції над файлами.

Перерахуємо основні функції файлової системи.

1. Ідентифікація файлів. Пов'язання імені файлу з виділеним йому простором зовнішньої пам'яті.

2. Розподіл зовнішньої пам'яті між файлами. Для роботи з конкретним файлом користувачеві не потрібно мати інформації про місцеположення цього файлу на зовнішньому носієві інформації. Наприклад, для того, щоб завантажити документ редактор з жорсткого диска, нам не потрібно знати, на якій стороні якого магнітного диска, на якому циліндрі і в якому секторі знаходиться даний документ.
3. Забезпечення надійності і відмовостійкої. Вартість інформації може у багато разів перевищувати вартість комп'ютера.
4. Забезпечення захисту від несанкціонованого доступу.
5. Забезпечення сумісного доступу до файлів, так щоб користувачеві не доводилося докладати спеціальних зусиль по забезпеченню синхронізації доступу.
6. Забезпечення високої продуктивності.

Іноді говорять, що файл - це пойменований набір зв'язаної інформації, записаної у вторинну пам'ять. Для більшості користувачів файлова система - найбільш видима частина ОС. Вона надає механізм для онлайнового зберігання і доступу як до даних, так і до програм для всіх користувачів системи. З погляду користувача, файл - одиниця зовнішньої пам'яті, тобто дані, записані на диск, мають бути у складі якого-небудь файлу.

Важливий аспект організації файлової системи - облік вартості операцій взаємодії з вторинною пам'яттю. Процес прочитування блоку диска складається з позиціонування прочитуючої головки над доріжкою, що містить необхідний блок, очікування, поки необхідний блок зробить оборот і опиниться під головкою, і власне прочитування блоку. Для цього потрібний значний час (десятки мілісекунд). У сучасних комп'ютерах звернення до диска здійснюється приблизно в 100 000 разів повільніше, ніж звернення до оперативної пам'яті. Таким чином, критерієм обчислювальної складності алгоритмів, що працюють із зовнішньою пам'яттю, є кількість звернень до диска.

У даній лекції розглядаються питання структури, іменування, захисту файлів; операції, які дозволяється проводити над файлами; організація файлового архіву (повного дерева довідників). Проблеми виділення дискового простору, забезпечення продуктивної роботи файлової системи і ряд інших питань, що цікавлять розробників системи, ви знайдете в наступній лекції.

Загальні відомості про файли

Імена файлів

Файлами є абстрактні об'єкти. Їх завдання - зберігати інформацію, приховуючи від користувача деталі роботи з пристроями. Коли процес створює файл, він дає йому ім'я. Після завершення процесу файл продовжує існувати і через своє ім'я може бути доступний іншим процесам.

Правила іменування файлів залежать від ОС. Багато ОС підтримують імена з двох частин (ім‘я+розширення), наприклад progr.c (файл, що містить текст програми на мові Сі) або autoexec.bat (файл, що містить команди інтерпретатора командної мови). Тип розширення файлу дозволяє ОС організувати роботу з ним різних прикладних програм відповідно до заздалегідь обумовлених угод. Зазвичай ОС накладають деякі обмеження, як на використовувані в імені символи, так і на довжину імені файлу. Відповідно до стандарту POSIX, популярні ОС оперують зручними для користувача довгими іменами (до 255 символів).

Типи файлів

Важливий аспект організації файлової системи і ОС - чи слід підтримувати і розпізнавати типи файлів. Якщо так, то це може допомогти правильному функціонуванню ОС, наприклад не допустити виводу на принтер бінарного файлу.

Основні типи файлів: регулярні (звичайні) файли і директорії (довідники, каталоги). Звичайні файли містять призначену для користувача інформацію. Директорії - системні файли, що підтримують структуру файлової системи. У каталозі міститься перелік вхідних в нього файлів і встановлюється відповідність між файлами і їх характеристиками (атрибутами). Ми розглядатимемо директорії нижче.

Нагадаємо, що хоча усередині підсистеми управління файлами звичайний файл представляється у вигляді набору блоків зовнішньої пам'яті, для користувачів забезпечується представлення файлу у вигляді лінійної послідовності байтів. Таке уявлення дозволяє використовувати абстракцію файлу при роботі із зовнішніми пристроями, при організації міжпроцесних взаємодій і так далі Так, наприклад, клавіатура зазвичай розглядається як текстовий файл, з якого комп'ютер отримує дані в символьному форматі. Тому іноді до файлів приписують інші об'єкти ОС, наприклад спеціальні символьні файли і спеціальні блокові файли, іменовані канали і сокети, що мають файловий інтерфейс. Ці об'єкти розглядаються в інших розділах даного курсу.

Далі мова піде головним чином про звичайні файли.

Звичайні (або регулярні) файли реально є набором блоків (можливо, порожній) на пристрої зовнішньої пам'яті, на якому підтримується файлова система. Такі файли можуть містити як текстову інформацію (зазвичай у форматі ASCII), так і довільну двійкову (бінарну) інформацію.

Текстові файли містять символьні рядки, які можна роздрукувати, побачити на екрані або редагувати звичайним текстовим редактором.

Інший тип файлів - нетекстові, або бінарні, файли. Зазвичай вони мають деяку внутрішню структуру. Наприклад, виконуваний файл в ОС Unix має п'ять секцій: заголовок, текст, дані, біти реаллокації і символьну таблицю. ОС виконує файл, тільки якщо він має потрібний формат. Іншим прикладом бінарного файлу може бути архівний файл. Типізація файлів не дуже строга.

Зазвичай прикладні програми, що працюють з файлами, розпізнають тип файлу по його імені відповідно до загальноприйнятих угод. Наприклад, файли з розширеннями .c, .pas, .txt - ASCII-файли, файли з розширеннями .exe - здійснимі, файли з розширеннями .obj, .zip - бінарні і так далі

Атрибути файлів

Окрім імені ОС часто пов'язують з кожним файлом і іншу інформацію, наприклад дату модифікації, розмір і так далі Ці інші характеристики файлів називаються атрибутами. Список атрибутів в різних ОС може варіюватися. Зазвичай він містить наступні елементи: основну інформацію (ім'я, тип файлу), адресну інформацію (пристрій, початкова адреса, розмір), інформацію про управління доступом (власник, допустимі операції) і інформацію про використання (дати створення, останнього читання, модифікації і ін.).

Список атрибутів зазвичай зберігається в структурі директорій (див. наступну лекцію) або інших структурах, що забезпечують доступ до даних файлу.

Організація файлів і доступ до них

Програміст сприймає файл у вигляді набору однорідних записів. Запис - це найменший елемент даних, який може бути оброблений як єдине ціле прикладною програмою при обміні із зовнішнім пристроєм. Причому в більшості ОС розмір запису дорівнює одному байту. Тоді як

застосування оперують записами, фізичний обмін з пристроєм здійснюється великими одиницями (зазвичай блоками). Тому записи об'єднуються в блоки для виводу і розблоковуються - для введення. Питання розподілу блоків зовнішньої пам'яті між файлами розглядаються в наступній лекції.

ОС підтримують декілька варіантів структуризації файлів.

Послідовний файл

Простий варіант - так званий послідовний файл. Тобто файл є послідовністю записів. Оскільки записи, як правило, однобайтові, файл є неструктурованою послідовністю байтів.

Обробка подібних файлів припускає послідовне читання записів від початку файлу, причому конкретний запис визначається її положенням у файлі. Такий спосіб доступу називається послідовним (модель стрічки). Якщо як носій файлу використовується магнітна стрічка, то так і робиться. Поточна позиція прочитування може бути повернена на початок файлу (rewind).

Файл прямого доступу

У реальній практиці файли зберігаються на пристроях прямого (random) доступу, наприклад на дисках, тому вміст файлу може бути розкидане по різних блоках диска, які можна прочитувати в довільному порядку. Причому номер блоку однозначно визначається позицією усередині файлу.

Тут мається на увазі відносний номер, що специфікує даний блок серед блоків диска, який належать файлу. Про зв'язок відносного номера блоку з абсолютним його номером на диску розповідається в наступній лекції.

В цьому випадку для доступу до середини файлу проглядання всього файлу із самого початку не обов'язкове. Для специфікації місця, з якого треба починати читання, використовуються два способи: з початку або з поточної позиції, яку дає операція seek. Файл, байти якого можуть бути лічені в довільному порядку, називається файлом прямого доступу.

Таким чином, файл, що складається з однобайтових записів на пристрої прямого доступу, - найбільш поширений спосіб організації файлу. Базовими операціями для такого роду файлів є прочитування або запис символу в поточну позицію. У більшості мов високого рівня передбачені оператори посимвольної пересилки даних у файл або з нього.

Подібну логічну структуру мають файли в багатьох файлових системах, наприклад у файлових системах ОС Unix і MS-DOS. ОС не здійснює ніякої інтерпретації вмісту файлу. Ця схема забезпечує максимальну гнучкість і універсальність. За допомогою базових системних викликів (або функцій бібліотеки введення/виводу) користувачі можуть як завгодно структурувати файли. Зокрема, багато СУБД зберігають свої бази даних в звичайних файлах.

Інші форми організації файлів

Відомі як інші форми організації файлу, так і інші способи доступу до них, які використовувалися в ранніх ОС, а також застосовуються сьогодні у великих мейнфреймах (mainframe), орієнтованих на комерційну обробку даних.

Перший крок в структуризації - зберігання файлу у вигляді послідовності записів фіксованої довжини, кожна з яких має внутрішню структуру. Операція читання проводиться над записом, а операція запису переписує або додає запис цілком. Раніше використовувалися записи по 80 байт (це відповідало числу позицій в перфокарті) або по 132 символи (ширина принтера). У ОС

CP/M файли були послідовностями 128-символьних записів. З введенням CRT-терміналів дана ідея втратила популярність.

Інший спосіб представлення файлів - послідовність записів змінної довжини, кожна з яких містить ключове поле у фіксованій позиції усередині запису (див. мал. 13.1). Базисна операція в даному випадку - прочитати запис з яким-небудь значенням ключа. Записи можуть розташовуватися у файлі послідовно (наприклад, відсортовані за значенням ключового поля) або в складнішому порядку. Метод доступу за значенням ключового поля до записів послідовного файлу називається індексно-послідовним.

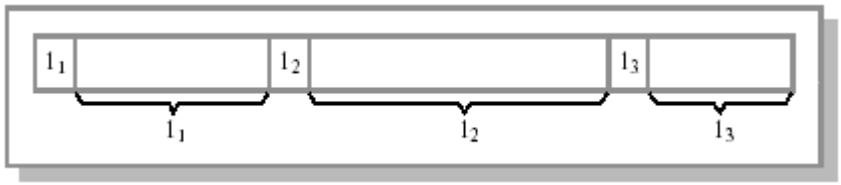


Рис. 13.1. Файл як послідовність записів змінної довжини

У деяких системах прискорення доступу до файлу забезпечується конструюванням індексу файлу. Індекс зазвичай зберігається на тому ж пристрої, що і сам файл, і складається із списку елементів, кожен з яких містить ідентифікатор запису, за яким слідує вказівка про місцеположення даного запису. Для пошуку запису спочатку відбувається звернення до індексу, де знаходиться покажчик на потрібний запис. Такі файли називаються індексованими, а метод доступу до них - доступ з використанням індексу.

Припустимо, у нас є великий несортований файл, що містить різноманітні відомості про студентів, що складаються із записів з декількома полями, і виникає завдання організації швидкого пошуку по одному з полів, наприклад по прізвищу студента. Мал. 13.2 ілюструє вирішення даної проблеми - організацію методу доступу до файлу з використанням індексу.

Рис. 13.2. Приклад організації індексу для послідовного файлу

Слід зазначити, що майже завжди головним чинником збільшення швидкості доступу є надмірність даних.

Спосіб виділення дискового простору за допомогою індексних вузлів, вживаний у ряді ОС (Unix і деяких інших, див. наступну лекцію), може служити іншим прикладом організації індексу.

В цьому випадку ОС використовує деревовидну організацію блоків, при якій блоки, складові файл, є листям дерева, а кожен внутрішній вузол містить покажчики на безліч блоків файлу.

Для великих файлів індекс може бути дуже великий. В цьому випадку створюють індекс для індексного файлу (блоки проміжного рівня або блоки непрямої адресації).

Операції над файлами

Операційна система повинна надати в розпорядження користувача набір операцій для роботи з файлами, реалізованих через системні виклики. Найчастіше при роботі з файлом користувач виконує не одну, а декілька операцій. По-перше, потрібно знайти дані файлу і його атрибути по символьному імені, по-друге, рахувати необхідні атрибути файлу у відведену область оперативної пам'яті і проаналізувати права користувача на виконання необхідної операції. Потім слід виконати операцію, після чого звільнити займану даними файлу область пам'яті. Розглянемо як приклад основні файлові операції ОС Unix [Таненбаум, 2002].

- Створення файлу, що не містить даних. Сенс даного виклику - оголосити, що файл існує, і привласнити йому ряд атрибутів. При цьому виділяється місце для файлу на диску і вноситься запис в каталог.
- Видалення файлу і звільнення займаного їм дискового простору.
- Відкриття файлу. Перед використанням файлу процес повинен його відкрити. Мета даного системного виклику - дозволити системі проаналізувати атрибути файлу і перевірити права доступу до нього, а також рахувати в оперативну пам'ять список адрес блоків файлу для швидкого доступу до його даних. Відкриття файлу є процедурою створення дескриптора або блоку файлу, що управляє. Дескриптор (описувач) файлу зберігає всю інформацію про нього. Іноді, відповідно до парадигми, прийнятої в мовах програмування, під дескриптором розуміється альтернативне ім'я файлу або покажчик на опис файлу в таблиці відкритих файлів, використовуваний при подальшій роботі з файлом. Наприклад, на мові С операція відкриття файлу fd=open(pathname,flags,modes); повертає дескриптор fd, який може бути задіяний при виконанні операцій читання (read(fd,buffer,count);) або запису.
- Закриття файлу. Якщо робота з файлом завершена, його атрибути і адреси блоків на диску більше не потрібні. В цьому випадку файл потрібно закрити, щоб звільнити місце у внутрішніх таблицях файлової системи.
- Позиціонування. Дає можливість специфікувати місце усередині файлу, звідки проводитиметься прочитування (або запис) даних, тобто задати поточну позицію.
- Читання даних з файлу. Зазвичай це робиться з поточної позиції. Користувач повинен задати об'єм прочитуваних даних і надати для них буфер в оперативній пам'яті.
- Запис даних у файл з поточної позиції. Якщо поточна позиція знаходиться в кінці файлу, його розмір збільшується, інакше запис здійснюється на місце наявних даних, які, таким чином, втрачаються.

Є і інші операції, наприклад перейменування файлу, отримання атрибутів файлу і так далі

Існує два способи виконати послідовність дій над файлами [Оліфер, 2001].

У першому випадку для кожної операції виконуються як універсальні, так і унікальні дії (схема stateless). Наприклад, послідовність операцій може бути такій: open, read1, close ... open, read2, close ... open, read3, close.

Альтернативний спосіб - це коли універсальні дії виконуються на початку і в кінці послідовності операцій, а для кожної проміжної операції виконуються тільки унікальні дії. В цьому випадку послідовність наведених вище операцій виглядатиме так: open, read1 ... read2 ... read3, close.

Більшість ОС використовує другий спосіб, економічніший і швидший. Перший спосіб стійкіший до збоїв, оскільки результати кожної операції стають незалежними від результатів попередньої операції; тому він іноді застосовується в розподілених файлових системах (наприклад, Sun NFS).

Директорії. Логічна структура файлового архіву

Кількість файлів на комп'ютері може бути великою. Окремі системи зберігають тисячі файлів, що займають сотні гігабайтів дискового простору. Ефективне управління цими даними має на увазі наявність в них чіткої логічної структури. Всі сучасні файлові системи підтримують багаторівневе іменування файлів за рахунок наявності в зовнішній пам'яті додаткових файлів із спеціальною структурою - каталогів (або директорій).

Кожен каталог містить список каталогів і/або файлів, що містяться в даному каталозі. Каталоги мають один і той же внутрішній формат, де кожному файлу відповідає один запис у файлі директорії (див., наприклад, рис.13.3).

Число директорій залежить від системи. У ранніх ОС була тільки одна коренева директорія, потім з'явилися директорії для користувачів (по одній директорії на користувача). У сучасних ОС використовується довільна структура дерева директорій.

Ім'я файлу (каталогу)	Тип файлу (звичайний чи каталог)	
Anti	K	атрибути
Games	K	атрибути
Autoexec.bat	O	атрибути
mouse.com	O	атрибути

Рис. 13.3. Директорії

Таким чином, файли на диску утворюють ієрархічну деревовидну структуру (див. мал. 13.4).

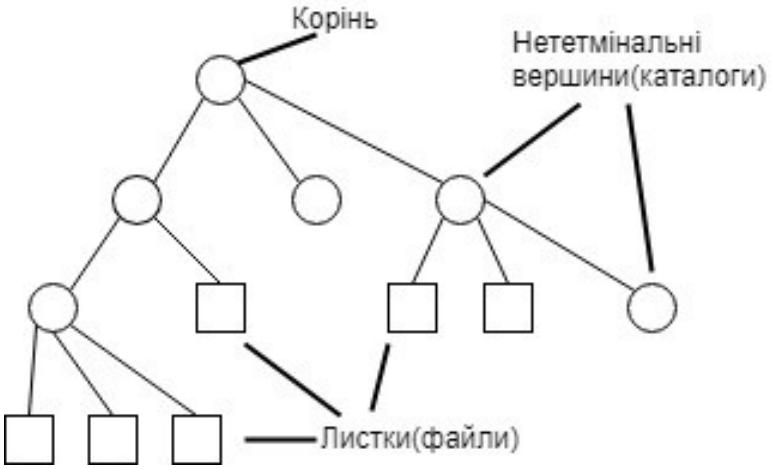


Рис. 13.4. Деревовидна структура файлової системи

Існує декілька еквівалентних способів зображення дерева. Структура перевернутого дерева, приведеного на мал. 13.4, найбільш поширена. Верхню вершину називають коренем. Якщо елемент дерева не може мати нащадків, він називається термінальною вершиною або листом (в даному випадку є файлом). Нелистові вершини - довідники або каталоги містять списки листових і нелистових вершин. Шлях від кореня до файлу однозначно визначає файл.

Подібні деревовидні структури є графами, що не мають циклів. Можна вважати, що ребра графа направлені вниз, а корінь - вершина, що не має вхідних ребер. Як ми побачимо в наступній лекції, скріплення файлів, яке практикується у ряді операційних систем, приводить до утворення циклів в графі.

Усередині одного каталогу імена листових файлів унікальні. Імена файлів, що знаходяться в різних каталогах, можуть збігатися. Для того, щоб однозначно визначити файл по його імені (уникнути колізії імен), прийнято іменувати файл так званим абсолютним або повним ім'ям (pathname), що складається із списку імен вкладених каталогів, по якому можна знайти шлях від кореня до файлу плюс ім'я файлу в каталозі, що безпосередньо містить даний файл. Тобто повне ім'я включає ланцюжок імен - шлях до файлу, наприклад /usr/games/doom. Такі імена унікальні. Компоненти шляху розділяють різними символами: "/" (слеш) в Unix або зворотним слешем в MS-DOS (у Multics - ">"). Таким чином, використання деревовидних каталогів мінімізує складність призначення унікальних імен.

Указувати повне ім'я не завжди зручно, тому застосовують інший спосіб завдання імені - відносний шлях до файлу. Він використовує концепцію робочої або поточної директорії, яка зазвичай входить до складу атрибутів процесу, що працює з даним файлом. Тоді на файли в такій директорії можна посилатися тільки по імені, при цьому пошук файлу здійснюватиметься в робочому каталозі. Це зручніше, але по суті, те ж саме, що і абсолютна форма.

Для діставання доступу до файлу і локалізації його блоків система повинна виконати навігацію по каталогах. Розглянемо для прикладу шлях /usr/linux/progr.c. Алгоритм однаковий для всіх ієрархічних систем. Спочатку у фіксованому місці на диску знаходиться коренева директорія. Потім знаходиться компонент шляху usr, тобто в кореневій директорії шукається файл /usr. Досліджуючи цей файл, система розуміє, що даний файл є каталогом, і блоки його даних розглядає як список файлів і шукає наступний компонент linux в ньому. З рядка для linux знаходиться файл, відповідний компоненту usr/linux/. Потім знаходиться компонент progr.c, який відкривається, заноситься в таблицю відкритих файлів і зберігається в ній до закриття файлу.

Відхилення від типової обробки компонентів pathname може виникнути у тому випадку, коли цей компонент є не звичайним каталогом з відповідним йому індексним вузлом і списком файлів, а служить точкою скріплення (прийнято говорити "точкою монтування") двох файлових архівів. Цей випадок розглянутий в наступній лекції.

Багато прикладних програм працюють з файлами, що знаходяться в поточній директорії, не указуючи явним чином її імені. Це дає користувачеві можливість довільним чином іменувати каталоги, що містять різні програмні пакети. Для реалізації цієї можливості в більшості ОС, що підтримують ієрархічну структуру директорій, використовується позначення "." - для поточної директорії і ".." - для батьківської.

Розділи диска. Організація доступу до архіву файлів.

Завдання шляху до файлу у файлових системах деяких ОС відрізняється тим, з чого починається цей ланцюжок імен.

У сучасних ОС прийнято розбивати диски на логічні диски (це низькорівнева операція), іноді звані розділами (partitions). Буває, що, навпаки, об'єднують декілька фізичних дисків в один логічний диск (наприклад, це можна зробити в ОС Windows NT). Тому в подальшому

викладі ми ігноруватимемо проблему фізичного виділення простору для файлів і вважатимемо, що кожним розділом є окремий (віртуальний) диск. Диск містить ієрархічну деревовидну структуру, що складається з набору файлів, кожен з яких є сховищем даних користувача, і каталогів або директорій (тобто файлів, які містять перелік інших файлів, що входять до складу каталогу), необхідних для зберігання інформації про файли системи.

У деяких системах управління файлами потрібний, щоб кожен архів файлів цілком розташовувався на одному диску (розділі диска). В цьому випадку повне ім'я файлу починається з імені дискового пристрою, на якому встановлений відповідний диск (букви диска). Наприклад, c:\util\nu\ndd.exe. Такий спосіб іменування використовується у файлових системах DEC і Microsoft.

У інших системах (Multics) вся сукупність файлів і каталогів є єдиним деревом. Сама система, виконуючи пошук файлів по імені, починаючи з кореня, вимагала установки необхідних дисків.

У ОС Unix передбачається наявність декількох архівів файлів, кожен на своєму розділі, один з яких вважається за кореневий. Після запуску системи можна "змонтувати" кореневу файлову систему і ряд ізольованих файлових систем в одну загальну файлову систему.

Технічно це здійснюється за допомогою створення в кореневій файловій системі спеціальних порожніх каталогів (див. також наступну лекцію). Спеціальний системний виклик mount ОС Unix дозволяє підключити до одного з цих порожніх каталогів кореневий каталог вказаного архіву файлів. Після монтування загальної файлової системи іменування файлів проводиться так само, як якби вона із самого початку була централізованою. Завданням ОС є безперешкодний прохід точки монтування при діставанні доступу до файлу по ланцюжку імен. Якщо врахувати, що звичайне монтування файлової системи проводиться при завантаженні системи, користувачі ОС Unix зазвичай і не замислюються про походження загальної файлової системи.

Операції над директоріями

Як і у випадку з файлами, система зобов'язана забезпечити користувача набором операцій, необхідних для роботи з директоріями, реалізованих через системні виклики. Не дивлячись на те що директорії - це файли, логіка роботи з ними відрізняється від логіки роботи із звичайними файлами і визначається природою цих об'єктів, призначених для підтримки структури файлового архіву. Сукупність системних викликів для управління директоріями залежить від особливостей конкретної ОС. Нагадаємо, що операції над каталогами є прерогативою ОС, тобто користувач не може, наприклад, виконати запис в каталог починаючи з поточної позиції. Розглянемо як приклад деякі системні виклики, необхідні для роботи з каталогами [Таненбаум, 2002].

- Створення директорії. Знову створена директорія включає записи з іменами '.' і '..', проте вважається порожній.
- Видалення директорії. Видалена може бути тільки порожня директорія.
- Відкриття директорії для подальшого читання. Наприклад, щоб перерахувати файли, що входять в директорію, процес повинен відкрити директорію і рахувати імена всіх файлів, які вона включає.
- Закриття директорії після її читання для звільнення місця у внутрішніх системних таблицях.

- Пошук. Даний системний виклик повертає вміст поточного запису у відкритій директорії. Взагалі кажучи, для цих цілей може використовуватися системний виклик Read, але в цьому випадку від програміста буде потрібно знання внутрішньої структури директорії.
- Отримання списку файлів в каталозі.
- Перейменування. Імена директорій можна міняти, як і імена файлів.
- Створення файлу. При створенні нового файлу необхідно додати в каталог відповідний елемент.
- Видалення файлу. Видалення з каталога відповідного елементу. Якщо файл, що видаляється, присутній тільки в одній директорії, то він взагалі віддаляється з файлової системи, інакше система обмежується тільки видаленням запису, що специфікується.

Очевидно, що створення і видалення файлів припускає також виконання відповідних файлових операцій. Є ще ряд інших системних викликів, наприклад пов'язаних із захистом інформації.

Захист файлів

Загальні проблеми безпеки ОС розглянуті в лекціях 15-16. Інформація в комп'ютерній системі має бути захищена як від фізичного руйнування (reliability), так і від несанкціонованого доступу (protection).

Тут ми торкнемося окремих аспектів захисту, пов'язаних з контролем доступу до файлів.

Контроль доступу до файлів

Наявність в системі багатьох користувачів припускає організацію контрольованого доступу до файлів. Виконання будь-якої операції над файлом має бути дозволене тільки у разі наявності у користувача відповідних привілеїв. Зазвичай контролюються наступні операції: читання, запис і виконання. Інші операції, наприклад копіювання файлів або їх перейменування, також можуть контролюватися. Проте вони частіше реалізуються через перерахованих. Так, операцію копіювання файлів можна представити як операцію читання і подальшу операцію запису.

Списки прав доступу

Найбільш загальний підхід до захисту файлів від несанкціонованого використання - зробити доступ залежним від ідентифікатора користувача, тобто пов'язати з кожним файлом або директорією список прав доступу (access control list), де перераховані імена користувачів і типи дозволених для них способів доступу до файлу. Будь-який запит на виконання операції звіряється з таким списком. Основна проблема реалізації даного способу - список може бути довгим. Щоб дозволити всім користувачам читати файл, необхідно всіх їх внести до списку. У такої техніки є два небажані слідства.

- Конструювання подібного списку може виявитися складним завданням, особливо якщо ми не знаємо заздалегідь користувачів системи.
- Запис в директорії повинен мати змінний розмір (включати список потенційних користувачів).

Для вирішення цих проблем створюють класифікації користувачів, наприклад, в ОС Unix всі користувачі розділені на три групи.

- Власник (Owner).

- Група (Group). Набір користувачів, що розділяють файл і потребують типового способу доступу до нього.
- Інші (Univers).

Це дозволяє реалізувати версію списку прав доступу, що конденсує. В рамках такої обмеженої класифікації задаються тільки три поля (поодинці для кожної групи) для кожної контрольованої операції. У результаті в Unix операції читання, записи і виконань контролюються за допомогою 9 бітів (rwxrwxrwx).

Висновок

Отже, файлова система є набором файлів, директорій і операцій над ними. Імена, структури файлів, способи доступу до них і їх атрибути - важливі аспекти організації файлової системи. Зазвичай файл є неструктурованою послідовністю байтів. Головне завдання файлової системи - пов'язати символьне ім'я файлу з даними на диску. Більшість сучасних ОС підтримують ієрархічну систему каталогів або директорій з можливим вкладенням директорій. Безпека файлової системи, що базується на веденні списків прав доступу, - одна з найважливіших концепцій ОС.

Як вже мовилося, файлова система повинна організувати ефективну роботу з даними, що зберігаються в зовнішній пам'яті, і надати користувачеві можливості для запам'ятовування і вибірки цих даних.

Для організації зберігання інформації на диску користувач спочатку зазвичай виконує його форматування, виділяючи на нім місце для структур даних, які описують стан файлової системи в цілому. Потім користувач створює потрібну йому структуру каталогів (або директорій), які, по суті, є списками вкладених каталогів і власне файлів. І нарешті, він заповнює дисковий простір файлами, приписуючи їх тому або іншому каталогу. Таким чином, ОС повинна надати в розпорядження користувача сукупність системних викликів, які забезпечують його необхідними сервісами.

Крім того, файлові служби можуть вирішувати проблеми перевірки і збереження цілісності файлової системи, проблеми підвищення продуктивності і ряд інших.

Загальна структура файлової системи

Система зберігання даних на дисках може бути структурована таким чином (див. мал. 14.1).

Нижній рівень - устаткування. Це насамперед магнітні диски з рухомими головками - основні пристрої зовнішньої пам'яті, що є пакетами магнітних пластин (поверхонь), між якими на одному важелі рухається пакет магнітних головок. Крок руху пакету головок є дискретним, і кожному положенню пакету головок логічно відповідає циліндр магнітного диска. Циліндри діляться на доріжки (треки), а кожна доріжка розмічається на одну і ту ж кількість блоків (секторів) таким чином, що в кожен блок можна записати по максимуму одне і те ж число байтів. Отже, для обміну з магнітним диском на рівні апаратури потрібно вказати номер циліндра, номер поверхні, номер блоку на відповідній доріжці і число байтів, яке потрібно записати або прочитати від початку цього блоку. Таким чином, диски можуть бути розбиті на блоки фіксованого розміру і можна безпосередньо дістати доступ до будь-якого блоку (організувати прямий доступ до файлів).

Безпосередньо з пристроями (дисками) взаємодіє частина ОС, звана системою введення-виводу. Система введення-виводу надає в розпорядження більш високорівневого компоненту ОС - файлової системи - використовуваний дисковий простір у вигляді безперервної послідовності блоків фіксованого розміру. Система введення-виводу має справу з фізичними блоками диска, які характеризуються адресою, наприклад диск 2, циліндр 75, сектор 11. Файлова система має справу з логічними блоками, кожен з яких має номер (від 0 або 1 до N). Розмір логічних блоків файлу збігається або є кратним розміру фізичного блоку диска і може бути заданий рівним розміру сторінки віртуальної пам'яті, підтримуваною апаратурою комп'ютера спільно з операційною системою.

У структурі системи управління файлами можна виділити базисну підсистему, яка відповідає за виділення дискового простору конкретним файлам, і більш високорівневу логічну підсистему, яка використовує структуру дерева директорій для надання модулю базисної підсистеми необхідної нею інформації, виходячи з символічного імені файлу. Вона також відповідальна за авторизацію доступу до файлів.

Стандартний запит на відкриття (open) або створення (create) файлу поступає від прикладної програми до логічної підсистеми. Логічна підсистема, використовуючи структуру директорій, перевіряє права доступу і викликає базову підсистему для дістування доступу до блоків файлу. Після цього файл вважається за відкритий, він міститься в таблиці відкритих файлів, і прикладна програма отримує в своє розпорядження дескриптор (або handle в системах Microsoft) цього файлу. Дескриптор файлу є посиланням на файл в таблиці відкритих

файлів і використовується в запитах прикладної програми на читання-запис з цього файлу. Запис в таблиці відкритих файлів указує через систему виділення блоків диска на блоки даного файлу. Якщо до моменту відкриття файл вже використовується іншим процесом, тобто міститься в таблиці відкритих файлів, то після перевірки прав доступу до файлу може бути організований сумісний доступ. При цьому новому процесу також повертається дескриптор - посилання на файл в таблиці відкритих файлів. Далі в тексті детально проаналізована робота найбільш важливих системних викликів.

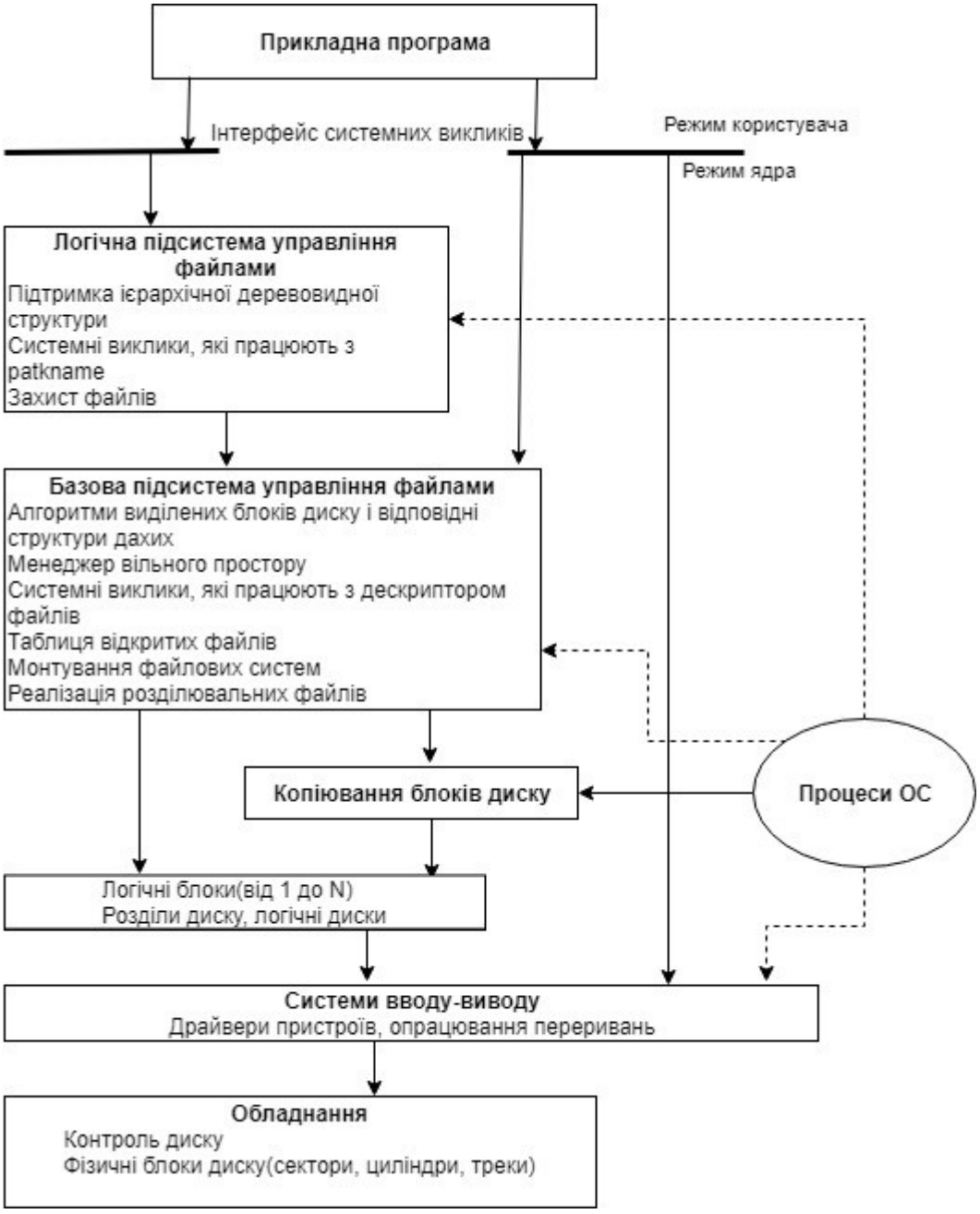


Рис. 14.1. Блок-схема файлової системи

Управління зовнішньою пам'яттю

Перш ніж описувати структуру даних файлової системи на диску, необхідно розглянути алгоритми виділення дискового простору і способи обліку вільної і зайнятої дискової пам'яті. Ці завдання зв'язані між собою.

Методи виділення дискового простору

Ключовим, безумовно, є питання, який тип структур використовується для обліку окремих блоків файлу, тобто спосіб скріплення файлів з блоками диска. У ОС використовується декілька методів виділення файлу дискового простору. Для кожного з методів запис в директорії, відповідний символьному імені файлу, містить покажчик, слідуючи якому можна знайти всі блоки даного файлу.

Виділення безперервною послідовністю блоків

Простий спосіб - зберігати кожен файл як безперервну послідовність блоків диска. При безперервному розташуванні файл характеризується адресою і довжиною (у блоках). Файл, що стартує з блоку b, займає потім блоки b+1, b+2 ... b+n-1.

Ця схема має дві переваги. По-перше, її легко реалізувати, оскільки з'ясування місцезнаходження файлу зводиться до питання, де знаходиться перший блок. По-друге, вона забезпечує хорошу продуктивність, оскільки цілий файл може бути лічений за одну дискову операцію.

Безперервне виділення використовується в ОС IBM/CMS, в ОС RSX-11 (для виконуваних файлів) і у ряді інших.

Цей спосіб поширений мало, і ось чому. В процесі експлуатації диск є деякою сукупністю вільних і зайнятих фрагментів. Не завжди є відповідний за розміром вільний фрагмент для нового файлу. Проблема безперервного розташування може розглядатися як окремий випадок більш загальної проблеми виділення блоку потрібного розміру із списку вільних блоків. Типовими рішеннями цієї задачі є стратегії першого відповідного, найбільш відповідного і найменш відповідного (порівняєте з проблемою виділення пам'яті в методі з динамічним розподілом). Як і у разі виділення потрібного об'єму оперативній пам'яті в схемі з динамічними розділами, метод страждає від зовнішньої фрагментації, більшою чи меншою мірою, залежно від розміру диска і середнього розміру файлу.

Крім того, безперервний розподіл зовнішньої пам'яті непридатний до тих пір, поки невідомий максимальний розмір файлу. Іноді розмір вихідного файлу оцінити легко (при копіюванні). Частіше, проте, це важко зробити, особливо в тих випадках, коли розмір файлу міняється. Якщо місця не вистачило, то призначена для користувача програма може бути припинена з урахуванням виділення додаткового місця для файлу при подальшому рестарті. Деякі ОС використовують модифікований варіант безперервного виділення - основні блоки файлу + резервні блоки. Проте з виділенням блоків з резерву виникають ті ж проблеми, оскільки доводиться вирішувати задачу виділення безперервній послідовності блоків диска тепер уже з сукупності резервних блоків.

Єдиним прийнятним вирішенням перерахованих проблем є періодичне ущільнення вмісту зовнішньої пам'яті, або "збірка сміття", мета якої полягає в об'єднанні вільних ділянок в один великий блок. Але це дорога операція, яку неможливо здійснювати дуже часто.

Таким чином, коли вміст диска постійно змінюється, даний метод нераціональний. Проте для стаціонарних файлових систем, наприклад для файлових систем компакт-дисків, він цілком придатний.

Зв'язний список

Зовнішня фрагментація - основна проблема розглянутого вище методу - може бути усунена за рахунок представлення файлу у вигляді зв'язного списку блоків диска. Запис в директорії містить покажчик на перший і останній блоки файлу (іноді як варіант використовується спеціальний знак кінця файлу - EOF). Кожен блок містить покажчик на наступний блок (див. мал. 14.2).

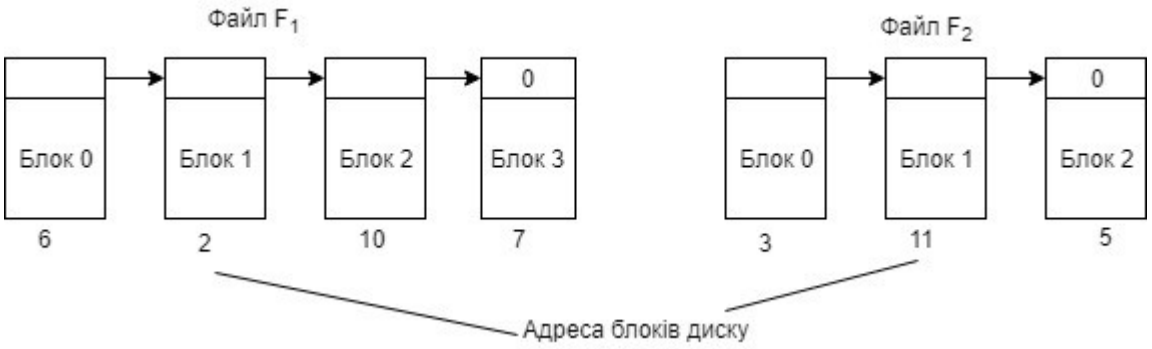


Рис. 14.2. Зберігання файлу у вигляді зв'язного списку дискових блоків

Зовнішня фрагментація для даного методу відсутня. Будь-який вільний блок може бути використаний для задоволення запиту. Відмітимо, що немає необхідності декларувати розмір файлу у момент створення. Файл може рости необмежено.

Зв'язне виділення має, проте, декілька істотних недоліків.

По-перше, при прямому доступі до файлу для пошуку i-го блоку потрібно здійснити декілька звернень до диска, послідовно прочитуючи блоки від 1 до i-1, тобто вибірка логічно суміжних записів, які займають фізично несуміжні сектори, може вимагати багато часу. Тут ми втрачаємо всі переваги прямого доступу до файлу.

По-друге, даний спосіб не дуже надійний. Наявність дефектного блоку в списку приводить до втрати інформації в частині файлу, що залишилася, і потенційно до втрати дискового простору, відведеного під цей файл.

Нарешті, для покажчика на наступний блок усередині блоку потрібно виділити місце, що не завжди зручно. Ємкість блоку, що традиційно є ступенем двійки (багато програм читають і пишуть блоками по ступенях двійки), таким чином, перестає бути ступенем двійки, оскільки покажчик відбирає декілька байтів.

Тому метод зв'язного списку зазвичай в чистому вигляді не використовується.

Таблиця відображення файлів

Одним з варіантів попереднього способу є зберігання покажчиків не в дискових блоках, а в індексній таблиці в пам'яті, яка називається таблицею відображення файлів (FAT - file allocation table) (див. мал. 14.3). Цієї схеми дотримуються багато ОС (MS-DOS, OS/2, MS Windows і ін.)

Як і раніше істотно, що запис в директорії містить тільки посилання на перший блок. Далі за допомогою таблиці FAT можна локалізувати блоки файлу незалежно від його розміру. У тих рядках таблиці, які відповідають останнім блокам файлів, зазвичай записується деяке граничне значення, наприклад EOF.

Головне достоїнство даного підходу полягає в тому, що по таблиці відображення можна судити про фізичне сусідство блоків, розташованих на диску, і при виділенні нового блоку можна легко знайти вільний блок диска, що знаходиться поблизу від інших блоків даного файлу. Мінусом даної схеми може бути необхідність зберігання в пам'яті цієї досить великої таблиці.

Номера блоків диску		
1		
2	10	
3	11	Початок файлу F_1
4		
5	EOF	
6	2	Початок файлу F_2
7	EOF	
8		
9		
10	7	
11	5	

Рис. 14.3. Метод зв'язного списку з використанням таблиці в оперативній пам'яті

Індексні вузли

Найбільш поширений метод виділення файлу блоків диска - пов'язати з кожним файлом невелику таблицю, звану індексним вузлом (i-node), яка перераховує атрибути і дискові адреси блоків файлу (див. рис 14.4). Запис в директорії, що відноситься до файлу, містить адресу індексного блоку. У міру заповнення файлу покажчики на блоки диска в індексному вузлі набувають осмислених значень.

Індексування підтримує прямий доступ до файлу, без збитку від зовнішньої фрагментації. Індексоване розміщення широко поширене і підтримує як послідовний, так і прямий доступ до файлу.

Зазвичай застосовується комбінація однорівневого і багаторівневих індексів. Перші декілька адрес блоків файлу зберігаються безпосередньо в індексному вузлі, таким чином, для маленьких файлів індексний вузол зберігає всю необхідну інформацію про адреси блоків диска. Для великих файлів одна з адрес індексного вузла указує на блок непрямої адресації. Даний блок містить адреси додаткових блоків диска. Якщо цього недостатньо, використовується блок подвійної непрямої адресації, який містить адреси блоків непрямої адресації. Якщо і це не вистачає, використовується блок потрійної непрямої адресації.

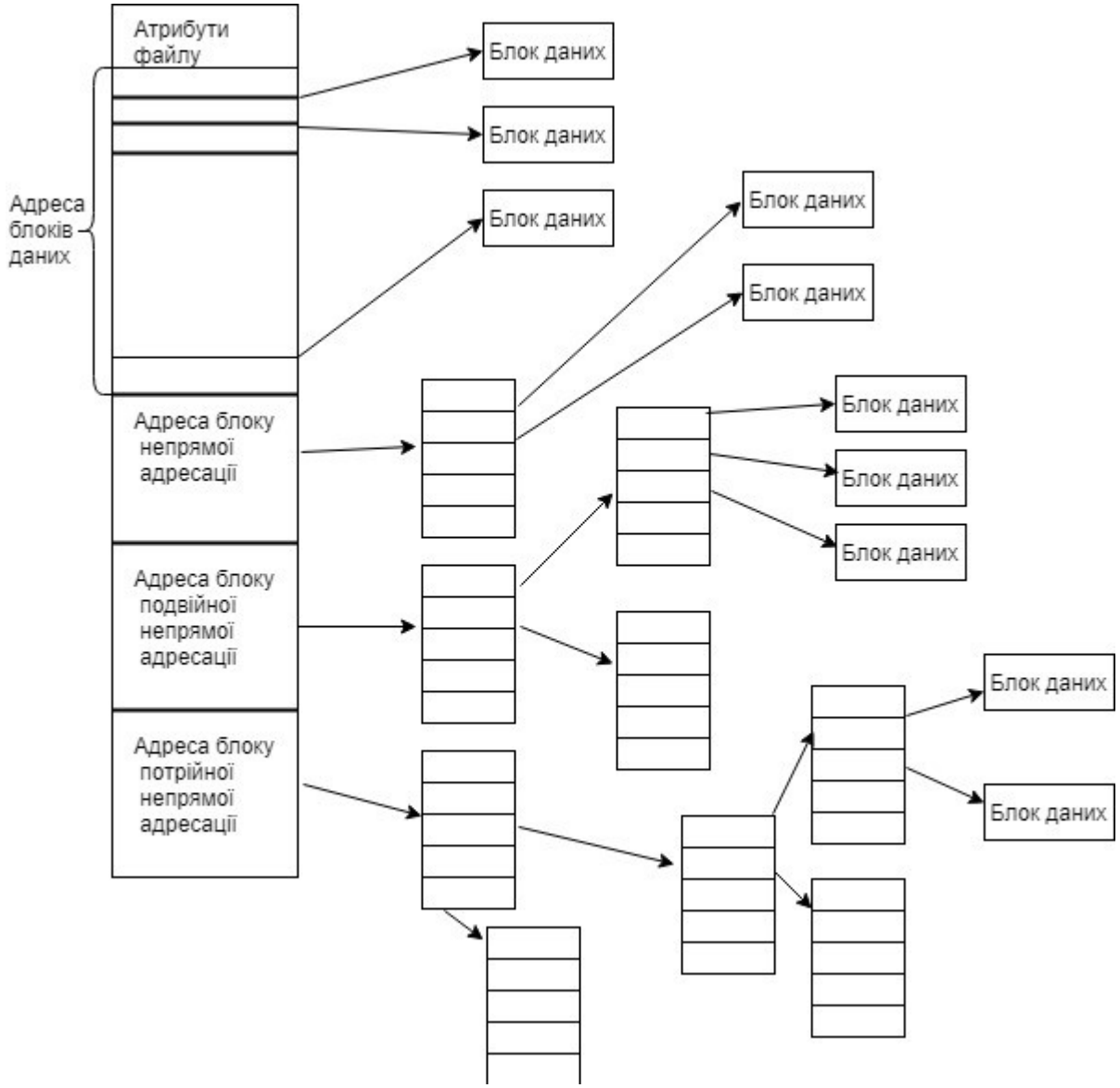


Рис. 14.4. Структура індексного вузла

Дану схему використовують файлові системи Unix (а також файлові системи HPFS, NTFS і ін.). Такий підхід дозволяє при фіксованому, відносно невеликому розмірі індексного вузла підтримувати роботу з файлами, розмір яких може мінятися від декількох байтів до декількох гігабайтів. Істотно, що для маленьких файлів використовується тільки пряма адресація, що забезпечує максимальну продуктивність.

Управління вільним і зайнятим дисковим простором

Дисковий простір, не виділений жодному файлу, також має бути керованим. У сучасних ОС використовується декілька способів обліку використовуваного місця на диску. Розглянемо найбільш поширені.

Облік за допомогою організації бітового вектора

Часто список вільних блоків диска реалізований у вигляді бітового вектора (bit map або bit vector). Кожен блок представлений одним бітом, що набуває значення 0 або 1, залежно від того, зайнятий він або вільний. Наприклад, 00111100111100011000001 ... .

Головна перевага цього підходу полягає в тому, що він відносно простий і ефективний при знаходженні першого вільного блоку або n послідовних блоків на диску. Багато комп'ютерів мають інструкції маніпулювання бітами, які можуть використовуватися для цієї мети.

Наприклад, комп'ютери сімейств Intel і Motorola мають інструкції, за допомогою яких можна легко локалізувати перший одиничний біт в слові.

Описуваний метод обліку вільних блоків використовується в Apple Macintosh.

Не дивлячись на те що розмір описаного бітового вектора найменший зі всіх можливих структур, навіть такий вектор може опинитися великого розміру. Тому даний метод ефективний, тільки якщо бітовий вектор поміщається в пам'яті цілком, що можливо лише для відносно невеликих дисків. Наприклад, диск розміром 4 Гбайт з блоками по 4 Кбайт потребує таблиці розміром 128 Кбайт для управління вільними блоками. Іноді, якщо бітовий вектор стає дуже великим, для прискорення пошуку в нім його розбивають на регіони і організовують резюмуючі структури даних, що містять зведення про кількість вільних блоків для кожного регіону.

Облік за допомогою організації зв'язного списку

Інший підхід - зв'язати в список всі вільні блоки, розміщуючи покажчик на перший вільний блок в спеціально відведеному місці диска, попутно кешуючи в пам'яті цю інформацію.

Подібна схема не завжди ефективна. Для трасування списку потрібно виконати багато звернень до диска. Проте, на щастя, нам необхідний, як правило, тільки перший вільний блок.

Іноді вдаються до модифікації підходу зв'язного списку, організовуючи зберігання адрес n вільних блоків в першому вільному блоці. Перші n-1 цих блоків дійсно використовуються. Останній блок містить адреси інших n блоків і так далі

Існують і інші методи, наприклад, вільний простір можна розглядати як файл і вести для нього відповідний індексний вузол.

Розмір блоку

Розмір логічного блоку грає важливу роль. У деяких системах (Unix) він може бути заданий при форматуванні диска. Невеликий розмір блоку приводитиме до того, що кожен файл міститиме багато блоків. Читання блоку здійснюється із затримками на пошук і обертання, таким чином, файл з багатьох блоків читатиметься поволі. Великі блоки забезпечують вищу швидкість обміну з диском, але із-за внутрішньої фрагментації (кожен файл займає ціле число блоків, і в середньому половина останнього блоку пропадає) знижується відсоток корисного дискового простору.

Для систем із сторінковою організацією пам'яті характерна схожа проблема з розміром сторінки.

Проведені дослідження показали, що більшість файлів мають невеликий розмір. Наприклад, в Unix приблизно 85% файлів мають розмір менше 8 Кбайт і 48% - менше 1 Кбайта.

Можна також врахувати, що в системах з віртуальною пам'яттю бажано, щоб одиницею пересилки диск-пам'ять була сторінка (найбільш поширений розмір сторінок пам'яті - 4 Кбайта). Звідси звичайний компромісний вибір блоку розміром 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт.

Структура файлової системи на диску

Розгляд методів роботи з дисковим простором дає загальне уявлення про сукупність службових даних, необхідних для опису файлової системи. Структура службових даних типової файлової системи, наприклад Unix, на одному з розділів диска, таким чином, може складатися з чотирьох основних частин (див. мал. 14.5).

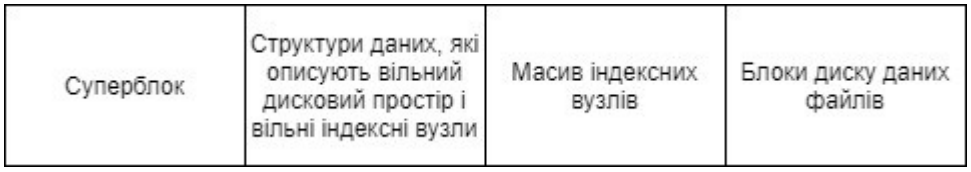


Рис. 14.5. Зразкова структура файлової системи на диску

На початку розділу знаходиться суперблок, що містить загальний опис файлової системи, наприклад:

- тип файлової системи;
- розмір файлової системи в блоках;
- розмір масиву індексних вузлів;
- розмір логічного блоку.

Описані структури даних створюються на диску в результаті його форматування (наприклад, утилітами format, makefs і ін.). Їх наявність дозволяє звертатися до даних на диску як до файлової системи, а не як до звичайної послідовності блоків.

У файлових системах сучасних ОС для підвищення стійкості підтримується декілька копій суперблоку. У деяких версіях Unix суперблок включав також і структури даних, керівники розподілом дискового простору, внаслідок чого суперблок безперервно піддавався модифікації, що знижувало надійність файлової системи в цілому. Виділення структур даних, що описують дисковий простір, в окрему частину є правильнішим рішенням.

Масив індексних вузлів (ilist) містить список індексів, відповідних файлам даної файлової системи. Розмір масиву індексних вузлів визначається адміністратором при установці системи. Максимальне число файлів, які можуть бути створені у файловій системі, визначається числом доступних індексних вузлів.

У блоках даних зберігаються реальні дані файлів. Розмір логічного блоку даних може задаватися при форматуванні файлової системи. Заповнення диска змістовною інформацією припускає використання блоків зберігання даних для файлів директорій і звичайних файлів і має слідством модифікацію масиву індексних вузлів і даних, що описують простір диска. Окремо узятий блок даних може належати одному і лише одному файлу у файловій системі.

Реалізація директорій

Як вже мовилося, директорія або каталог - це файл, що має вид таблиці і зберігаючий список вхідних в нього файлів або каталогів. Основне завдання файлів-директорій - підтримка ієрархічної деревовидної структури файлової системи. Запис в директорії має визначений для даної ОС формат, часто невідомий користувачеві, тому блоки даних файлу-директорії заповнюються не через операції запису, а за допомогою спеціальних системних викликів (наприклад, створення файлу).

Для доступу до файлу ОС використовує шлях (pathname), повідомлений користувачем. Запис в директорії зв'язує ім'я файлу або ім'я піддиректорії з блоками даних на диску (див. мал. 14.6). Залежно від способу виділення файлу блоків диска (див. розділ "Методи виділення дискового простору") це посилання може бути номером першого блоку або номером індексного вузла. У будь-якому випадку забезпечується зв'язок символьного імені файлу з даними на диску.

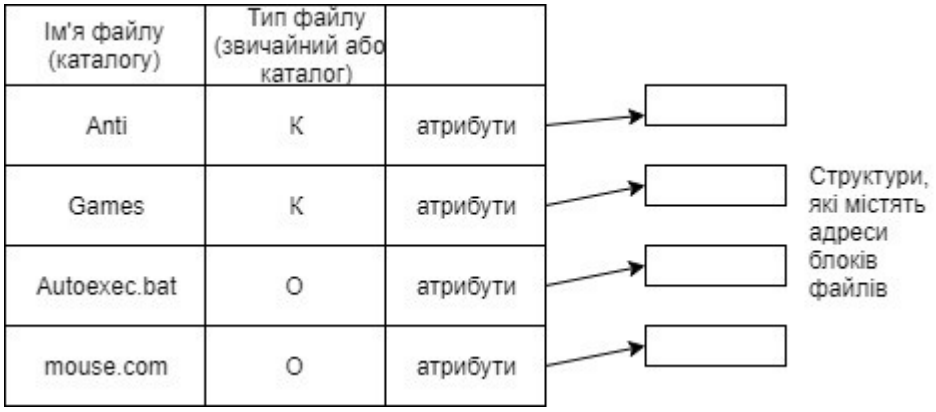


Рис. 14.6. Реалізація директорій

Коли система відкриває файл, вона шукає його ім'я в директорії. Потім із запису в директорії або із структури, на який запис в директорії указує, витягуються атрибути і адреси блоків файлу на диску. Ця інформація поміщається в системну таблицю в головній пам'яті. Все подальші посилання на даний файл використовують цю інформацію. Атрибути файлу можна зберігати безпосередньо в записі в директорії, як показано на мал. 14.6. Проте для організації сумісного доступу до файлів зручніше зберігати атрибути в індексному вузлі, як це робиться в Unix.

Розглянемо декілька конкретних прикладів.

Приклади реалізації директорій в деяких ОС

Директорії в ОС MS-DOS

У ОС MS-DOS типовий запис в директорії має вигляд, показаний на мал. 14.7.

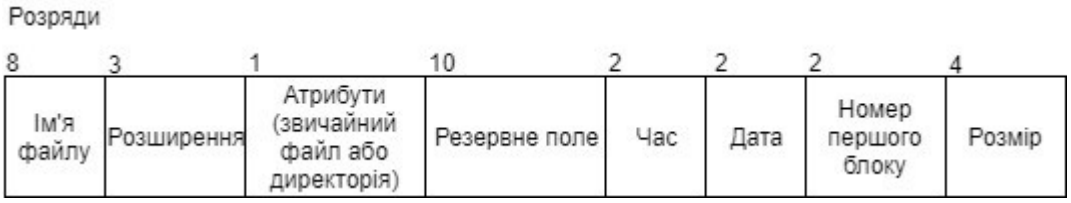


Рис. 14.7. Варіант запису в директорії MS-DOS

У ОС MS-DOS, як і в більшості сучасних ОС, директорії можуть містити піддиректорії (що специфікуються бітом атрибуту), що дозволяє конструювати довільне дерево директорій файлової системи.

Номер першого блоку використовується як індекс в таблиці FAT. Далі по ланцюжку в цій таблиці може бути знайдені решта блоків.

Директорії в ОС Unix

Структура директорії проста. Кожен запис містить ім'я файлу і номер його індексного вузла (див. мал. 14.8). Решта всієї інформації про файл (тип, розмір, час модифікації, власник і так далі і номери дискових блоків) знаходиться в індексному вузлі.

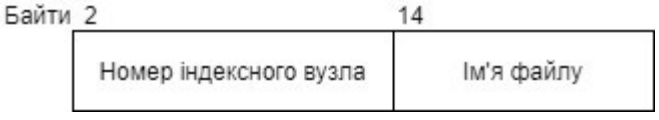


Рис. 14.8. Варіант запису в директорії Unix

У пізніших версіях Unix форма запису зазнала ряд змін, наприклад ім'я файлу описується структурою. Проте суть залишилася колишньою.

Пошук в директорії

Список файлів в директорії зазвичай не є впорядкованим по іменах файлів. Тому правильний вибір алгоритму пошуку імені файлу в директорії має великий вплив на ефективність і надійність файлових систем.

Лінійний пошук

Існує декілька стратегій проглядання списку символьних імен. Простим з них є лінійний пошук. Директорія є видимою із самого початку, поки не зустрінеться потрібне ім'я файлу. Хоча це найменш ефективний спосіб пошуку, виявляється, що в більшості випадків він працює з прийнятною продуктивністю. Наприклад, автори Unix стверджували, що лінійного пошуку цілком достатньо. Мабуть, це пов'язано з тим, що на тлі відносного повільного доступу до диска деякі затримки, що виникають в процесі сканування списку, неістотні.

Метод простий, але вимагає тимчасових витрат. Для створення нового файлу спочатку потрібно перевірити директорію на наявність такого ж імені. Потім ім'я нового файлу вставляється в кінець директорії (якщо, зрозуміло, файл з таким же ім'ям в директорії не існує, інакше потрібно інформувати користувача). Для видалення файлу потрібно також виконати пошук його імені в списку і помітити запис як невживану.

Реальний недолік даного методу - послідовний пошук файлу. Інформація про структуру директорії використовується часто, і неефективний спосіб пошуку буде помітний користувачами. Можна звести пошук до бінарного, якщо відсортувати список файлів. Проте це ускладнить створення і видалення файлів, оскільки потрібне переміщення великого об'єму інформації.

Хеш-кодування-таблиця

Хешування (див. наприклад [Ахо, 2001]) - інший спосіб, який може використовуватися для розміщення і подальшого пошуку імені файлу в директорії. У даному методі імена файлів також зберігаються в каталозі у вигляді лінійного списку, але додатково використовується хеш-кодування-таблиця. Хеш-кодування-таблиця, точніше побудована на її основі хеш-кодування-функція, дозволяє по імені файлу отримати покажчик на ймення файл в списку. Таким чином, можна істотно зменшити час пошуку.

В результаті хешування можуть виникати колізії, тобто ситуації, коли функція хешування, застосована до різних імен файлів, дає один і той же результат. Зазвичай імена таких файлів об'єднують в зв'язкові списки, припускаючи надалі здійснення в них послідовного пошуку потрібного імені файлу. Вибір відповідного алгоритму хешування дозволяє звести до мінімуму число колізій. Проте завжди є вірогідність несприятливого результату, коли непропорційно великому числу імен файлів функція хешування ставить у відповідність один і той же результат. У такому разі перевага використання цієї схеми в порівнянні з послідовним пошуком практично втрачається.

Інші методи пошуку

Окрім описаних методів пошуку імені файлу, в директорії існують та інші. Як приклад можна привести організацію пошуку в каталогах файлової системи NTFS за допомогою так званого B-дерева, яке стало стандартним способом організації індексів в системах баз даних (див. [Ахо, 2001]).

Монтування файлових систем

Так само як файл має бути відкритий перед використанням, і файлова система, що зберігається на розділі диска, має бути змонтована, щоб стати доступною процесам системи.

Функція mount (вмонтовувати) зв'язує файлову систему з вказаного розділу на диску з існуючою ієрархією файлових систем, а функція umount (демонтувати) вимикає файлову систему з ієрархії. Функція mount, таким чином, дає користувачам можливість звертатися до даних в дисковому розділі як до файлової системи, а не як до послідовності дискових блоків.

Процедура монтування полягає в наступному. Користувач (у Unix це суперкористувач) повідомляє ОС ім'я пристрою і місце у файловій структурі (ім'я порожнього каталогу), куди потрібно приєднати файлову систему (точка монтування) (див. мал. 14.9 і мал. 14.10). Наприклад, в ОС Unix бібліотечний виклик mount має вигляд:

mount(special pathname,directory pathname,options);

де special pathname - ім'я спеціального файлу пристрою (у загальному випадку ім'я розділу), відповідного дисковому розділу з вмонтовуваною файловою системою, directory pathname - каталог в існуючій ієрархії, де вмонтовуватиметься файлова система (іншими словами, крапка або місце монтування), а options вказує, чи слід вмонтовувати файлову систему "тільки для читання" (при цьому не виконуватимуться такі функції, як write і creat, які проводять запис у файлову систему). Потім ОС повинна переконатися, що пристрій містить дійсну файлову систему очікуваного формату з суперблоком, списком індексів і кореневим індексом.

Деякі ОС здійснюють монтування автоматично, як тільки зустрінуть диск вперше (жорсткі диски на етапі завантаження, гнучкі, - коли вони вставлені в дисковод), ОС шукає файлову систему на пристрої. Якщо файлова система на пристрої є, вона вмонтовується на кореневому рівні, при цьому до ланцюжка імен абсолютного імені файлу (pathname) додається буква розділу.

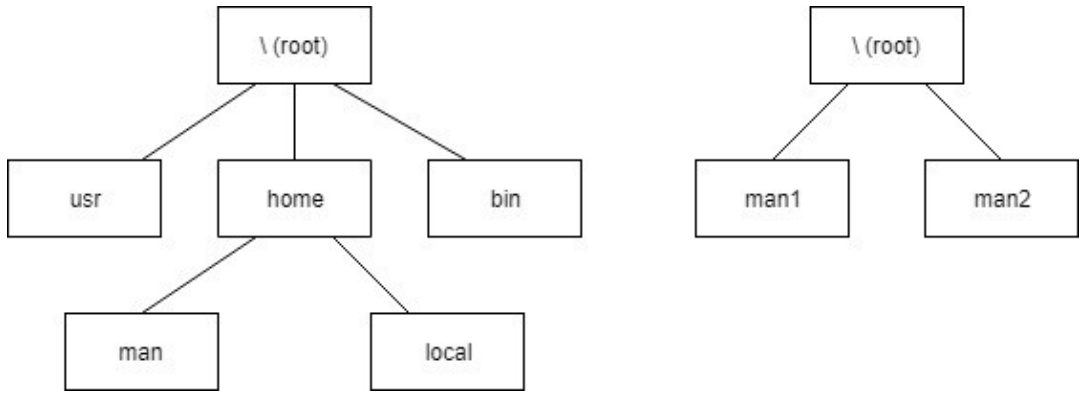


Рис. 14.9. Дві файлові системи до монтування

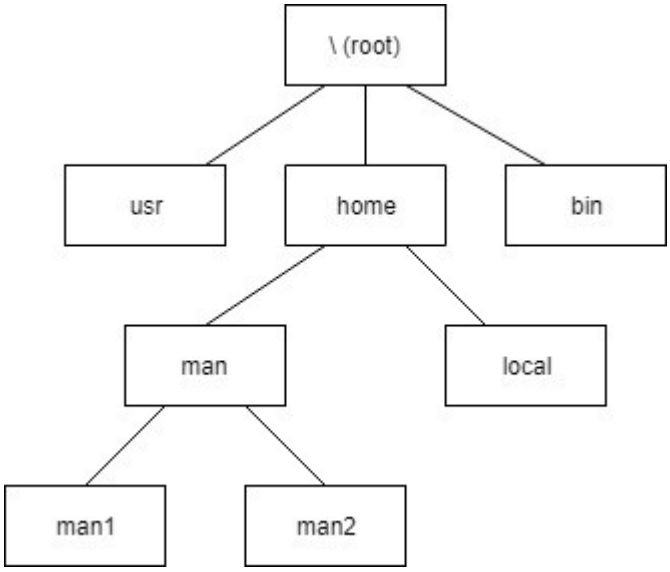


Рис. 14.10. Загальна файлова система після монтування

Ядро підтримує таблицю монтування із записами про кожну змонтовану файлову систему. У кожному записі міститься інформація про знов змонтований пристрій, про його суперблок і кореневий каталог, а також зведення про точку монтування. Для усунення потенційно небезпечних побічних ефектів число лінків (див. наступний розділ) до каталогу - точки монтування - має дорівнювати 1. Занесення інформації в таблицю монтування проводиться негайно, оскільки може виникнути конфлікт між двома процесами. Наприклад, якщо вмонтовуючий процес припинений для відкриття пристрою або прочитування суперблоку файлової системи, а тим часом інший процес може спробувати змонтувати файлову систему.

Наявність в логічній структурі файлового архіву точок монтування вимагає акуратної реалізації алгоритмів, що здійснюють навігацію по каталогах. Точку монтування можна перетнути двома способами: з файлової системи, де проводиться монтування, у файлову систему, яка вмонтовується (у напрямі від глобального кореня до листа), і у зворотному напрямі. Алгоритми пошуку файлів повинні передбачати ситуації, в яких черговий компонент шляху до файлу є точкою монтування, коли замість аналізу індексного вузла чергової директорії доводиться здійснювати обробку суперблоку вмонтовуваної системи.

Скріплення файлів

Ієрархічна організація, покладена в основу деревовидної структури файлової системи сучасних ОС, не передбачає виразу стосунків, в яких нащадки зв'язуються більш ніж з одним предком. Така негнучкість частково усувається можливістю реалізації скріплення файлів або організації линков (link).

Ядро дозволяє користувачеві зв'язувати каталоги, спрощуючи написання програм, що вимагають перетину дерева файлової системи (див. мал. 14.11). Часто має сенс зберігати під різними іменами одну і ту ж команду (виконуваний файл). Наприклад, виконуваний файл традиційного текстового редактора ОС Unix vi зазвичай може викликатися під іменами ex, edit, vi, view і vedit файлової системи. З'єднання між директорією і файлом, що розділяється, називається "зв'язком" або "посиланням" (link). Дерево файлової системи перетворюється на циклічний граф.

Це зручно, але створює ряд додаткових проблем.

Простий спосіб реалізувати скріплення файлу - просто дублювати інформацію про нього в обох директоріях. При цьому, проте, може виникнути проблема сумісності у випадку, якщо

власники цих директорій спробують незалежно один від одного змінити вміст файлу. Наприклад, в ОС CP/M запис в директорії про файл безпосередньо містить адреси дискових блоків. Тому копії тих же дискових адрес мають бути зроблені і в іншій директорії, куди файл лінкується. Якщо один з користувачів щось додає до файлу, нові блоки будуть перераховані тільки у нього в директорії і не будуть "видимі" іншому користувачеві.

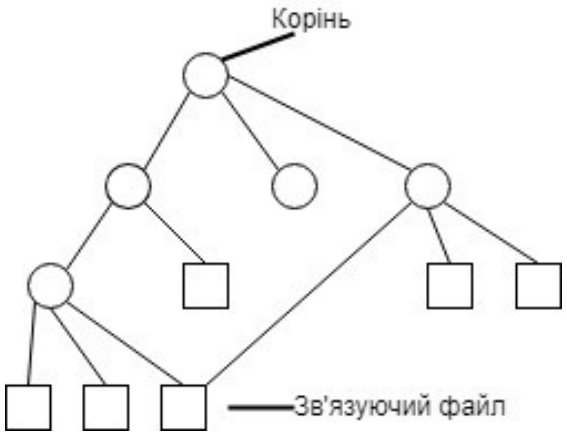


Рис. 14.11. Структура файлової системи з можливістю зв'язування файлу з новим іменем

Проблема такого роду може бути вирішена двома способами. Перший з них - так званий жорсткий зв'язок (hard link). Якщо блоки даних файлу перераховані не в директорії, а в невеликій структурі даних (наприклад, в індексному вузлі), пов'язаній власне з файлом, то другий користувач може зв'язатися безпосередньо з цією, вже існуючою структурою.

Альтернативне рішення - створення нового файлу, який містить шлях до зв'язуваного файлу. Такий підхід називається символічним лінковим (soft або symbolic link). При цьому у відповідному каталозі створюється елемент, в якому імені зв'язку зіставляється деяке ім'я файлу (цей файл навіть не зобов'язаний існувати до моменту створення символічного зв'язку). Для символічного зв'язку може створюватися окремий індексний вузол і навіть заводитися окремий блок даних для зберігання потенційно довгого імені файлу.

Кожен з цих методів має свої мінуси. У разі жорсткого зв'язку виникає необхідність підтримки лічильника посилань на файл для коректної реалізації операції видалення файлу. Наприклад, в Unix такий лічильник є одним з атрибутів, що зберігаються в індексному вузлі. Видалення файлу одним з користувачів зменшує кількість посилань на файл на 1. Реальне видалення файлу відбувається, коли число посилань на файл стає рівним 0.

У разі символічного лінкування така проблема не виникає, оскільки тільки реальний власник має посилання на індексний вузол файлу. Якщо власник видаляє файл, то він руйнується, і спроби інших користувачів працювати з ним закінчаться провалом. Видалення символічного линка на файл ніяк не впливає. Проблема організації символічного зв'язку - потенційне зниження швидкості доступу до файлу. Файл символічного линка зберігає шлях до файлу, що містить список вкладених директорій, для проходження по якому необхідно здійснити декілька звернень до диска.

Символічний лінк має ту перевагу, що він може використовуватися для організації зручного доступу до файлів видалених комп'ютерів, якщо, наприклад, додати до шляху мережеву адресу видаленої машини.

Циклічний граф - структура гнучкіша, ніж просте дерево, але робота з нею вимагає великої акуратності. Оскільки тепер до файлу існує декілька шляхів, програма пошуку файлу може знайти його на диску кілька разів. Просте практичне вирішення даної проблеми - обмежити число директорій при пошуку. Повне усунення циклів при пошуку - досить трудомістка

процедура, що виконується спеціальними утилітами і пов'язана з багатократним трасуванням директорій файлової системи.

Кооперація процесів при роботі з файлами

Коли різні користувачі працюють разом над проектом, вони часто потребують розділення файлів.

Файл, що розділяється, - ресурс, що розділяється. Як і у разі будь-якого спільно використовуваного ресурсу, процеси повинні синхронізувати доступ до спільно використовуваних файлів, каталогів, щоб уникнути тупикових ситуацій, дискримінації окремих процесів і зниження продуктивності системи.

Наприклад, якщо декілька користувачів одночасно редагують який-небудь файл і не прийнято спеціальних заходів, то результат буде непередбачуваний і залежить від того, в якому порядку здійснювалися записи у файл. Між двома операціями read одного процесу інший процес може модифікувати дані, що для багатьох застосувань неприйнятно. Просте вирішення даної проблеми - надати можливість одному з процесів захопити файл, тобто блокувати доступ до файлу інших процесів, що розділяється, на весь час, поки файл залишається відкритим для даного процесу. Проте це було б недостатньо гнучко і не відповідало б характеру поставленого завдання.

Розглянемо спочатку грубий підхід, тобто тимчасове захоплення призначеним для користувача процесом файлу або запису (частини файлу між вказаними позиціями).

Системний виклик, що дозволяє встановити і перевірити блокування на файл, є невід'ємним атрибутом сучасних багатокористувацьких ОС. В принципі, було б логічно зв'язати синхронізацію доступу до файлу як до єдиного цілого з системним викликом open (т. е., наприклад, відкриття файлу в режимі запису або оновлення могло б означати його монопольне блокування відповідним процесом, а відкриття в режимі читання - сумісне блокування). Так вчиняють в багатьох операційних системах (починаючи з ОС Multics). У ОС Unix це не так, що має історичні причини.

У першій версії системи Unix, розробленої Томпсоном і Річі, механізм захоплення файлу був відсутній. Застосовувався дуже простий підхід до забезпечення паралельного (від декількох процесів) доступу до файлів: система дозволяла будь-якому числу процесів одночасно відкривати один і той же файл в будь-якому режимі (читання, записи або оновлення) і не робила ніяких дій синхронізацій. Вся відповідальність за коректність сумісної обробки файлу лягала на процеси, що використовують його, і система навіть не надавала яких-небудь особливих засобів для синхронізації доступу процесів до файлу. Проте згодом для того, щоб підвищити привабливість системи для комерційних користувачів, що працюють з базами даних, у версію V системи було включено механізми захоплення файлу і записи, що базуються на системному виклику fcntl.

Допускається два варіанти синхронізації: з очікуванням, коли вимога блокування може привести до відкладання процесу до того моменту, коли ця вимога може бути задоволене, і без очікування, коли процес негайно оповіщається про задоволення вимоги блокування або про неможливість її задоволення в даний момент.

Встановлені блокування відносяться тільки до того процесу, який їх встановив, і не успадковуються процесами-нащадками цього процесу. Більш того, навіть якщо деякий процес користується можливостями синхронізацій системного виклику fcntl, інші процеси як і раніше можуть працювати з тим файлом без всякої синхронізації. Іншими словами, це справа групи процесів, що спільно використовують файл, - домовитися про спосіб синхронізації паралельного доступу.

Тонший підхід полягає в прозорому для користувача блокуванні окремих структур ядра, що відповідають за роботу з файлами частини призначених для користувача даних. Наприклад, в ОС Unix під час системного виклику, що здійснює ту або іншу операцію з файлом, як правило, відбувається блокування індексного вузла, що містить адреси блоків даних файлу. Може здатися, що організація блокувань або заборони більш ніж одному процесу працювати з файлом під час виконання системного виклику є зайвою, оскільки в переважній більшості випадків виконання системних викликів і так не уривається, тобто ядро працює в умовах невитісняючої багатозадачності. Проте в даному випадку це не зовсім так. Операції читання і запису займають тривалий час і лише ініціюються центральним процесором, а здійснюються по незалежних каналах, тому установка блокувань на час системного виклику є необхідною гарантією атомарності операцій читання і запису. На практиці виявляється достатнім заблокувати один з буферів кеша диска, в заголовку якого ведеться список процесів, чекаючих звільнення даного буфера. Таким чином, відповідно до семантики Unix зміни, зроблені одним користувачем, негайно стають "видні" іншому користувачеві, який тримає даний файл відкритим одночасно з першим.

Приклади дозволу колізій і тупикових ситуацій

Логіка роботи системи в складних ситуаціях може проілюструвати особливості організації мультидоступу.

Розглянемо як приклад утворення потенційної безвиході при створенні зв'язку (link), коли дозволений сумісний доступ до файлу [Bach, 1986].

Два процеси, наступні функції, що виконують одночасно:

процес А: link("a/b/c/d","e/f/g");

процес В: link("e/f","a/b/c/d/ee");

можуть зайти в безвихідь. Припустимо, що процес А виявив індекс файлу "a/b/c/d" в той самий момент, коли процес В виявив індекс файлу "e/f". Фраза "в той же момент" означає, що системою досягнутий стан, при якому кожен процес отримав шуканий індекс. Коли ж тепер процес А спробує отримати індекс файлу "e/f", він припинить своє виконання до тих пір, поки індекс файлу "f" не звільниться. В той же час процес В намагається отримати індекс каталогу "a/b/c/d" і припиняється в очікуванні звільнення індексу файлу "d". Процес А утримуватиме заблокованим індекс, потрібний процесу В, а процес В, у свою чергу, утримуватиме заблокованим індекс, необхідний процесу А.

Для запобігання цьому класичному прикладу взаємного блокування у файловій системі прийнято, щоб ядро звільняло індекс початкового файлу після збільшення значення лічильника зв'язків. Тоді, оскільки перший з ресурсів (індекс) вільний при зверненні до наступного ресурсу, взаємного блокування не відбувається.

Приводів для небажаної конкуренції між процесами багато, особливо при видаленні імен каталогів. Припустимо, що один процес намагається знайти дані файлу по його повному символічному імені, послідовно проходячи компонент за компонентом, а інший процес видаляє каталог, ім'я якого входить в шлях пошуку. Допустимо, процес А робить розбір імені "a/b/c/d" і припиняється під час отримання індексного вузла для файлу "c". Він може припинитися при спробі заблокувати індексний вузол або при спробі звернутися до дискового блоку, де цей індексний вузол зберігається. Якщо процесу В потрібно видалити зв'язок для каталога з ім'ям "c", він може припинитися з тієї ж самої причини, що і процес А. Хай ядро згодом вирішить відновити процес В раніше процесу А. Перш ніж процес А продовжить своє виконання, процес В завершиться, видаливши зв'язок каталогу "c" і його вміст по цьому зв'язку. Пізніше процес А спробує звернутися до неіснуючого індексного вузла, який вже був видалений. Алгоритм

пошуку файлу, перевіряючий насамперед нерівність значення лічильника зв'язків> нулю, повинен повідомити про помилку.

Можна привести і інші приклади, які демонструють необхідність ретельного проектування файлової системи для її подальшої надійної роботи.

Надійність файлової системи

Життя повне неприємних несподіванок, а руйнування файлової системи часто небезпечніше, ніж руйнування комп'ютера. Тому файлові системи повинні розроблятися з урахуванням подібної можливості. Окрім очевидних рішень, наприклад своєчасне дублювання інформації (backup), файлові системи сучасних ОС містять спеціальні засоби для підтримки власної сумісності.

Цілісність файлової системи

Важливий аспект надійної роботи файлової системи - контроль її цілісності. В результаті файлових операцій блоки диска можуть прочитуватися в пам'ять, модифікуватися і потім записуватися на диск. Причому багато файлових операцій зачіпають відразу декілька об'єктів файлової системи. Наприклад, копіювання файлу припускає виділення йому блоків диска, формування індексного вузла, зміна вмісту каталогу і так далі Протягом короткого періоду часу між цими кроками інформація у файловій системі опиняється неузгодженою.

І якщо наслідок непередбачуваної зупинки системи на диску будуть збережені зміни тільки для частини цих об'єктів (порушена атомарність файлової операції), файлова система на диску може бути залишена в несумісному стані. В результаті можуть виникнути порушення логіки роботи з даними, наприклад з'явитися "втрачені" блоки диска, які не належать жодному файлу і в той же час помічені як зайняті, або, навпаки, блоки, помічені як вільні, але в той же час зайняті (на них є посилання в індексному вузлі) або інші порушення.

У сучасних ОС передбачені заходи, які дозволяють звести до мінімуму збиток від псування файлової системи і потім повністю або частково відновити її цілісність.

Порядок виконання операцій

Очевидно, що для правильного функціонування файлової системи значущість окремих даних нерівноцінна. Спотворення вмісту призначених для користувача файлів не приводить до серйозних (з погляду цілісності файлової системи) наслідків, тоді як невідповідності у файлах, що містять інформацію (директорії, індексні вузли, суперблок і т. п.), що управляє, можуть бути катастрофічними. Тому має бути ретельно продуманий порядок виконання операцій із структурами даних файлової системи.

Розглянемо приклад створення жорсткого зв'язку для файлу [Робачевський, 1999]. Для цього файловій системі необхідно виконати наступні операції:

- створити новий запис в каталозі, вказуючий на індексний вузол файлу;
- збільшити лічильник зв'язків в індексному вузлі.

Якщо аварійна зупинка відбувся між 1-ою і 2-ою операціями, то в каталогах файлової системи існуватимуть два імена файлу, що адресують індексний вузол із значенням лічильника зв'язків, рівному 1. Якщо тепер буде видалено одне з імен, це приведе до видалення файлу як такого. Якщо ж порядок операцій змінений і, як раніше, зупинка відбулася між першою і другою операціями, файл матиме неіснуючий жорсткий зв'язок, але існуючий запис в каталозі буде правильним. Хоча це теж є помилкою, але її наслідки менш серйозні, ніж у попередньому випадку.

Журналізація

Іншим засобом підтримки цілісності є запозичений з систем управління базами даних прийом, званий журналізація (іноді вживається термін "журналювання"). Послідовність дій з об'єктами під час файлової операції протоколюється, і якщо відбулася зупинка системи, то, маючи в наявності протокол, можна здійснити відкіт системи назад в початковий цілісний стан, в якому вона перебувала до початку операції. Подібна надмірність може коштувати дорого, але вона виправдана, оскільки у разі відмови дозволяє реконструювати втрачені дані.

Для відкоту необхідно, щоб для кожної протокольованої в журналі операції існувала зворотна. Наприклад, для каталогів і реляційних СУБД це саме так. З цієї причини, на відміну від СУБД, у файлових системах протоколюються не всі зміни, а лише зміни метаданих (індексних вузлів, записів в каталогах і ін.). Зміни в даних користувача в протокол не заносяться. Крім того, якщо протоколювати зміни призначених для користувача даних, то цим буде завдано серйозного збитку продуктивності системи, оскільки кешування втратить сенс.

Журналізація реалізована в NTFS, Ext3FS, REISERFS і інших системах. Щоб підкреслити складність завдання, потрібно відзначити, що існують не цілком очевидні проблеми, пов'язані з процедурою відкоту. Наприклад, відміна одних змін може зачіпати дані, вже використані іншими файловими операціями. Це означає, що такі операції також мають бути скасовані. Дана проблема отримала назву каскадного відкоту транзакцій [Брукшир, 2001]

Перевірка цілісності файлової системи за допомогою утиліт

Якщо ж порушення все ж таки відбулося, то для усунення проблеми несумісності можна вдатися до утиліт (fsck, chkdsk, scandisk і ін.), які перевіряють цілісність файлової системи. Вони можуть запускатися після завантаження або після збою і здійснюють багатократне сканування різноманітних структур даних файлової системи у пошуках суперечностей.

Можливі також евристичні перевірки. Наприклад, знаходження індексного вузла, номер якого перевищує їх число на диску або пошук в призначених для користувача директоріях файлів, що належать суперкористувачеві.

На жаль, доводиться констатувати, що не існує ніяких засобів, що гарантують абсолютне збереження інформації у файлах, і в тих ситуаціях, коли цілісність інформації потрібно гарантувати з високим ступенем надійності, удаються до дорогих процедур дублювання.

Управління "поганими" блоками

Наявність дефектних блоків на диску - звичайна справа. Усередині блоку разом з даними зберігається контрольна сума даних. Під "поганими" блоками зазвичай розуміють блоки диску, для яких обчислена контрольна сума даних, які прочитуються, не збігається з контрольною сумою, що зберігається. Дефектні блоки зазвичай з'являються в процесі експлуатації. Іноді вони вже є при постачанні разом із списком, оскільки дуже скрутно для постачальників зробити диск повністю вільним від дефектів. Розглянемо два вирішення проблеми дефектних блоків - одне на рівні апаратури, інше на рівні ядра ОС.

Перший спосіб - зберігати список поганих блоків в контроллері диска. Коли контроллер ініціалізувався, він читає погані блоки і заміщає дефектний блок резервним, позначаючи відображення в списку поганих блоків. Всі реальні запити йтимуть до резервного блоку. Слід мати на увазі, що при цьому механізм підйомника (найбільш поширений механізм обробки запитів до блоків диска) працюватиме неефективно. Річ у тому, що існує стратегія черговості обробки запитів до диска (докладніше за див. лекцію "введення-виведення"). Стратегія диктує напрям руху прочитуючої головки диска до потрібного циліндра. Зазвичай резервні блоки розміщуються на зовнішніх циліндрах. Якщо поганий блок розташований на внутрішньому

циліндрі і контроллер здійснює підстановку прозорим чином, то рух головки, що здається, здійснюватиметься до внутрішнього циліндра, а фактичне - до зовнішнього. Це є порушенням стратегії і, отже, мінусом даної схеми.

Рішення на рівні ОС може бути наступним. Перш за все, необхідно ретельно сконструювати файл, що містить дефектні блоки. Тоді вони вилучаються із списку вільних блоків. Потім потрібно якимсь чином приховати цей файл від прикладних програм.

Продуктивність файлової системи

Оскільки звернення до диска - операція відносно повільна, мінімізація кількості таких звернень - ключове завдання всіх алгоритмів, що працюють із зовнішньою пам'яттю. Найбільш типова техніка підвищення швидкості роботи з диском - кешування.

Кешування

Кешем диска є буфер в оперативній пам'яті, що містить ряд блоків диска (див. мал. 14.12). Якщо є запит на читання/запис блоку диска, то спочатку проводиться перевірка на предмет наявності цього блоку в кеші. Якщо блок в кеші є, то запит задовольняється з кеша, інакше запитаний блок прочитується в кеш з диска. Скорочення кількості дискових операцій виявляється можливим унаслідок властивої ОС властивості локальності (про властивість локальності багато мовилося в лекціях, присвячених опису роботи системи управління пам'яттю).

Акуратна реалізація кешування вимагає вирішення декількох проблем.

По-перше, ємкість буфера кеша обмежена. Коли блок має бути завантажений в заповнений буфер кеша, виникає проблема заміщення блоків, тобто окремі блоки мають бути видалені з нього. Тут працюють ті ж стратегії і ті ж FIFO, Second Chance і LRU-алгоритми заміщення, що і при виштовхуванні сторінок пам'яті.

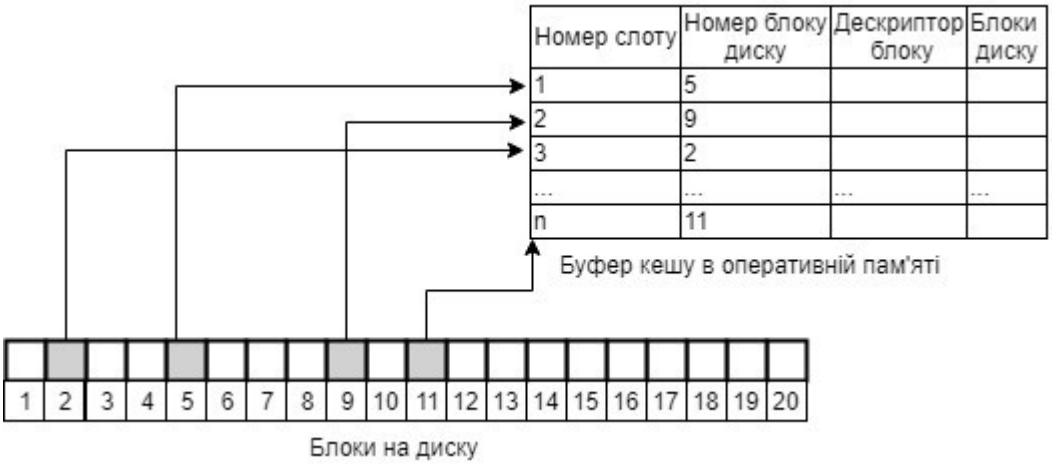


Рис. 14.12. Структура блокового кешу

Заміщення блоків повинне здійснюватися з урахуванням їх важливості для файлової системи. Блоки мають бути розділені на категорії, наприклад: блоки індексних вузлів, блоки непрямої адресації, блоки директорій, заповнені блоки даних і т. д., і залежно від приналежності блоку до тієї або іншої категорії можна застосовувати до них різну стратегію заміщення.

По-друге, оскільки кешування використовує механізм відкладеного запису, при якому модифікація буфера не викликає негайного запису на диск, серйозною проблемою є "старіння" інформації в дискових блоках, образи яких знаходяться в буферному кеші. Невчасна синхронізація буфера кеша і диска може привести до дуже небажаних наслідків у разі відмов

устаткування або програмного забезпечення. Тому стратегія і порядок відображення інформації з кеша на диск має бути ретельно продумана.

Так, блоки, істотні для сумісності файлової системи (блоки індексних вузлів, блоки непрямої адресації, блоки директорій), мають бути переписані на диск негайно, незалежно від того, в якій частині LRU-ланцюжку вони знаходяться. Необхідно ретельно вибрати порядок такого переписування.

У Unix є для цього виклик SYNC, який примушує всі модифіковані блоки записуватися на диск негайно. Для синхронізації вмісту кеша і диска періодично запускається фоновий процес-демон. Крім того, можна організувати синхронний режим роботи з окремими файлами, що задається при відкритті файлу, коли всі зміни у файлі негайно зберігаються на диску.

Нарешті, проблема конкуренції процесів на доступ до блоків кеша вирішується веденням списків блоків, що перебувають в різних станах, і відміткою про полягання блоку в його дескрипторі. Наприклад, блок може бути заблокований, брати участь в операції введення-виводу, а також мати список процесів, які чекають звільнення даного блоку.

Оптимальне розміщення інформації на диску

Кешування - не єдиний спосіб збільшення продуктивності системи. Інша важлива техніка - скорочення кількості рухів прочитуючої головки диску за рахунок розумної стратегії розміщення інформації. Наприклад, масив індексних вузлів в Unix прагнуть розмістити на середніх доріжках. Також має сенс розміщувати індексні вузли поблизу від блоків даних, на які вони посилаються і так далі

Крім того, рекомендується періодично здійснювати дефрагментацію диску (збірку сміття), оскільки в популярних методиках виділення дискових блоків (за виключенням, можливо, FAT) принцип локальності не працює, і послідовна обробка файлу вимагає звернення до різних ділянок диска.

Сучасна архітектура файлових систем

Сучасні ОС надають користувачеві можливість працювати відразу з декількома файловими системами (Linux працює з Ext2fs, FAT і ін.). Файлова система в традиційному розумінні стає частиною більш загальної багаторівневої структури (див. мал. 14.13).

На верхньому рівні розташовується так званий диспетчер файлових систем (наприклад, в Windows 95 цей компонент називається installable filesystem manager). Він пов'язує запити прикладної програми з конкретною файловою системою.

Кожна файлова система (іноді говорять - драйвер файлової системи) на етапі ініціалізації реєструється у диспетчера, повідомляючи йому точки входу, для подальших звернень до даної файлової системи.

Та ж ідея підтримки декількох файлових систем в рамках однієї ОС може бути реалізована по-іншому, наприклад виходячи з концепції віртуальної файлової системи. Віртуальна файлова система (vfs) є незалежним від реалізації рівнем і спирається на реальні файлові системи (s5fs, ufs, FAT, NFS, FFS. Ext2fs s). При цьому виникають структури даних віртуальної файлової системи типу віртуальних індексних вузлів vnode, які узагальнюють індексні вузли конкретних систем.

Мал. 14.13. Архітектура сучасної файлової системи

Висновок

Реалізація файлової системи пов'язана з такими питаннями, як підтримка поняття логічного блоку диска, скріплення імені файлу і блоків його даних, проблемами розділення файлів і проблемами управління дискового простору.

Найбільш поширені способи виділення дискового простору: безперервне виділення, організація зв'язного списку і система з індексними вузлами.

Файлова система часто реалізується у вигляді листкової модульної структури. Нижні шари мають справу з устаткуванням, а верхні - з символічними іменами і логічними властивостями файлів.

Директорії можуть бути організовані різними способами і можуть зберігати атрибути файлу і адреси блоків файлів, а іноді для цього призначається спеціальна структура (індексні вузли).

Проблеми надійності і продуктивності файлової системи – найважливіші аспекти її дизайну.

День 9.

Модуль 5. Управління введенням–виведенням в операційних системах.

Функціонування будь-якої обчислювальної системи зазвичай зводиться до виконання двох видів роботи: обробці інформації і операцій по здійсненню її введення-виводу. Оскільки в рамках моделі, прийнятої в даному курсі, все, що виконується в обчислювальній системі, організовано як набір процесів, ці два види роботи виконуються процесами. Процеси займаються обробкою інформації і виконанням операцій введення-виводу.

Зміст понять "Обробка інформації" і "операції введення-виводу" залежить від того, з якої точки зору ми дивимося на них. З погляду програміста, під "обробкою інформації" розуміється виконання команд процесора над даними, лежачими в пам'яті незалежно від рівня ієрархії, – в регістрах, кеші, оперативній або вторинній пам'яті. Під "операціями введення-виводу" програміст розуміє обмін даними між пам'яттю і пристроями, зовнішніми по відношенню до пам'яті і процесора, такими як магнітні стрічки, диски, монітор, клавіатура, таймер. З погляду операційної системи "обробкою інформації" є тільки операції, що здійснюються процесором над даними, що знаходяться в пам'яті на рівні ієрархії не нижче, ніж оперативна пам'ять. Все останнє відноситься до "операцій введення-виводу". Щоб виконувати операції над даними, тимчасово розташованими у вторинній пам'яті, операційна система спочатку проводить їх підкачку в оперативну пам'ять, і лише потім процесор здійснює необхідні дії.

Пояснення того, що саме робить процесор при обробці інформації, як він вирішує задачу і який алгоритм виконує, не входить в завдання нашого курсу. Це швидше відноситься до курсу "Алгоритми і структури даних", з якого зазвичай починається вивчення інформатики. Як операційна система управляє обробкою інформації, ми розібрали в частині II, в деталях описавши два стани процесів – виконання (а що його описувати те?) і готовності (черги планування і т. д.), а також правила, по яких здійснюється переклад процесів з одного стану в інше (алгоритми планування процесів).

Дана лекція буде присвячена другому виду роботи обчислювальної системи – операціям введення-виводу. Ми розберемо, що відбувається в комп'ютері при виконанні операцій введення-виводу, і як операційна система управляє їх виконанням. При цьому для простоти вважатимемо, що об'єм оперативної пам'яті в обчислювальній системі достатньо великий, тобто всі процеси повністю розташовуються в оперативній пам'яті, і тому поняття "Операція введення-виводу" з погляду операційної системи і з погляду користувача означає одне і те ж. Таке припущення не знижує спільності нашого розгляду, оскільки підкачка інформації з вторинної пам'яті в оперативну пам'ять і назад зазвичай будується за тим же принципом, що і всі операції введення-виводу.

Перш ніж говорити про роботу операційної системи при здійсненні операцій введення-виводу, нам доведеться пригадати деякі відомості з курсу "Архітектура сучасних ЕОМ і мова Асемблера", щоб зрозуміти, як здійснюється передача інформації між оперативною пам'яттю і зовнішнім пристроєм і чому для підключення до обчислювальної системи нових пристроїв її не потрібно перепроектувати.

Фізичні принципи організації введення-виводу

Існує багато різноманітних пристроїв, які можуть взаємодіяти з процесором і пам'яттю: таймер, жорсткі диски, клавіатура, дисплеї, миша, модеми і т. д., аж до пристроїв відображення і введення інформації в авіаційно-космічних тренажерах. Частина цих пристроїв може бути вбудована всередину корпусу комп'ютера, частина – винесена за його межі і спілкуватися з комп'ютером через різні лінії зв'язку: кабельні, оптоволоконні, радіорелейні, супутникові і так

далі Конкретний набір пристроїв і способи їх підключення визначаються цілями функціонування обчислювальної системи, бажаннями і фінансовими можливостями користувача. Не дивлячись на все різноманіття пристроїв, управління їх роботою і обмін інформацією з ними будуються на відносно невеликому наборі принципів, які ми постараємося розібрати в цьому розділі.

Загальні відомості про архітектуру комп'ютера

У простому випадку процесор, пам'ять і численні зовнішні пристрої зв'язані великою кількістю електричних з'єднань – ліній, які в сукупності прийнято називати локальною магістраллю комп'ютера. Усередині локальної магістралі лінії, службовці для передачі схожих сигналів і призначені для виконання схожих функцій, прийнято групувати в шини. При цьому поняття шини включає не тільки набір провідників, але і набір жорстко заданих протоколів, що визначає перелік повідомлень, який може бути переданий за допомогою електричних сигналів по цих провідниках. У сучасних комп'ютерах виділяють як мінімум три шини:

- шину даних, що складається з ліній даних і службовку для передачі інформації між процесором і пам'яттю, процесором і пристроями введення-виводу, пам'яттю і зовнішніми пристроями;
- адресну шину, що складається з ліній адреси і службовку для завдання адреси елементу пам'яті або вказівки пристрою введення-виводу, що беруть участь в обміні інформацією;
- шину управління, що складається з ліній управління локальною магістраллю і ліній її стану, що визначають поведінку локальної магістралі. У деяких архітектурних вирішеннях лінії стану виносяться з цієї шини в окрему шину стану.

Кількість ліній, що входять до складу шини, прийнято називати розрядністю (шириною) цієї шини. Ширина адресної шини, наприклад, визначає максимальний розмір оперативної пам'яті, яка може бути встановлена в обчислювальній системі. Ширина шини даних визначає максимальний об'єм інформації, яка за один раз може бути отримана або передана по цій шині.

Операції обміну інформацією здійснюються при одночасній участі всіх шин. Розглянемо, наприклад, дії, які мають бути виконані для передачі інформації з процесора в пам'ять. У простому випадку необхідно виконати три дії.

1. На адресній шині процесор повинен виставити сигнали, відповідні адресі елементу пам'яті, в яку здійснюватиметься передача інформації.
2. На шину даних процесор повинен виставити сигнали, відповідні інформації, яка має бути записана в пам'ять.
3. Після виконання дій 1 і 2 на шину управління виставляються сигнали, відповідні операції запису і роботі з пам'яттю, що приведе до занесення необхідної інформації за потрібною адресою.

Природно, що приведені вище дії є необхідними, але недостатніми при розгляді роботи конкретних процесорів і мікросхем пам'яті. Конкретні архітектурні рішення можуть вимагати додаткових дій: наприклад, виставляння на шину управління сигналів часткового використання шини даних (для передачі меншої кількості інформації, чим дозволяє ширина цієї шини); виставляння сигналу готовності магістралі після завершення запису в пам'ять, що дозволяє приступити до нової операції, і так далі Проте загальні принципи виконання операції запису в пам'ять залишаються незмінними.

Тоді як пам'ять легко можна уявити собі у вигляді послідовності пронумерованих адресами осередків, локалізованих усередині однієї мікросхеми або набору мікросхем, до пристроїв

введення-виводу подібний підхід непридатний. Зовнішні пристрої рознесені просторово і можуть підключатися до локальної магістралі в одній крапці або безлічі крапок, що отримали назву портів введення-виводу. Проте, точно так, як і елементи пам'яті взаємно однозначно відображалися в адресний простір пам'яті, порти введення-виводу можна взаємно однозначно відобразити в інший адресний простір – адресний простір введення-виводу. При цьому кожен порт введення-виводу отримує свій номер або адреса в цьому просторі. В деяких випадках, коли адресний простір пам'яті (розмір якого визначається шириною адресної шини) задіяний не повністю (залишилися адреси, яким не відповідають фізичні елементи пам'яті) і протоколи роботи із зовнішнім пристроєм сумісні з протоколами роботи з пам'яттю, частина портів введення -виводу може бути відображена безпосередньо в адресний простір пам'яті (так, наприклад, поступають з відеопам'яттю дисплеїв), правда, тоді ці порти вже не прийнято називати портами. Треба відзначити, що при відображенні портів в адресний простір пам'яті для організації доступу до них повною мірою можуть бути задіяні існуючі механізми захисту пам'яті без організації спеціальних захисних пристроїв.

У ситуації прямого відображення портів введення-виводу в адресний простір пам'яті дії, необхідні для запису інформації і команд, що управляють, в ці порти або для читання даних з них і їх станів, нічим не відрізняються від дій, вироблюваних для передачі інформації між оперативною пам'яттю і процесором, і для їх виконання застосовуються ті ж самі команди. Якщо ж порт відображений в адресний простір введення-виводу, то процес обміну інформацією ініціюється спеціальними командами введення-виводу і включає дещо інші дії. Наприклад, для передачі даних в порт необхідно виконати наступне.

- На адресній шині процесор повинен виставити сигнали, відповідні адресі порту, в який здійснюватиметься передача інформації, в адресному просторі введення-виводу.
- На шину даних процесор повинен виставити сигнали, відповідні інформації, яка має бути передана в порт.
- Після виконання дій 1 і 2 на шину управління виставляються сигнали, відповідні операції запису і роботі з пристроями введення-виводу (перемикання адресних просторів!), що приведе до передачі необхідній інформації в потрібний порт.

Істотна відмінність пам'яті від пристроїв введення-виводу полягає в тому, що занесення інформації в пам'ять є закінченням операції запису, тоді як занесення інформації в порт часто є ініціалізацією реального здійснення операції введення-виводу. Що саме повинні робити пристрої, прийнявши інформацію через свій порт, і яким саме образом вони повинні поставляти інформацію для читання з порту, визначається електронними схемами пристроїв, що отримали назву контроллерів. Контроллер може безпосередньо управляти окремим пристроєм (наприклад, контроллер диску), а може управляти декількома пристроями, зв'язуючись з їх контроллерами за допомогою спеціальних шин введення-виводу (шина IDE, шина SCSI і т. д.).

Сучасні обчислювальні системи можуть мати різноманітну архітектуру, безліч шин і магістралей, мости для переходу інформації від однієї шини до іншої і тому подібне Для нас зараз важливими є тільки наступні моменти.

- Пристрої введення-виводу підключаються до системи через порти.
- Можуть існувати два адресні простори: простір пам'яті і простір введення-виводу.
- Порти, як правило, відображаються в адресний простір введення-виводу і іноді – безпосередньо в адресний простір пам'яті.
- Використання того або іншого адресного простору визначається типом команди, що виконується процесором, або типом її операндів.

- Фізичним управлінням пристроєм введення-виводу, передачею інформації через порт і виставлянням деяких сигналів на магістралі займається контроллер пристрою.

Саме одноманітність підключення зовнішніх пристроїв до обчислювальної системи є одній з складових ідеології, що дозволяють додавати нові пристрої без того, що перепроектувало всієї системи.

Структура контроллера пристрою

Контроллери пристроїв введення-виводу вельми різні як по своїй внутрішній будові, так і по виконання (від однієї мікросхеми до спеціалізованої обчислювальної системи зі своїм процесором, пам'яттю і т. д.), оскільки їм доводиться управляти абсолютно різними приладами. Не вдаючись до деталей цих відмінностей, ми виділимо деякі загальні риси контроллерів, необхідні їм для взаємодії з обчислювальною системою. Зазвичай кожен контроллер має принаймні чотири внутрішні регістри, званих регістрами стану, управління, вхідних даних і вихідних даних. Для доступу до вмісту цих регістрів обчислювальна система може використовувати один або декілька портів, що для нас не істотно. Для простоти викладу вважатимемо, що кожному регістру відповідає свій порт.

Регістр стану містить біти, значення яких визначається станом пристрою введення-виводу і які доступні тільки для читання обчислювальною системою. Ці біти відображають завершення виконання поточної команди на пристрої (біт зайнятості), наявність чергового даного в регістрі вихідних даних (біт готовності даних), виникнення помилки при виконанні команди (біт помилки) і так далі

Регістр управління отримує дані, які записуються обчислювальною системою для ініціалізації пристрою введення-виводу або виконання чергової команди, а також зміни режиму роботи пристрою. Частина бітів в цьому регістрі може бути відведена під код виконуваної команди, частина бітів кодуватиме режим роботи пристрою, біт готовності команди свідчить про те, що можна приступити до її виконання.

Регістр вихідних даних служить для приміщення в нього даних для читання обчислювальною системою, а регістр вхідних даних призначений для приміщення в нього інформації, яка має бути виведена на пристрій. Зазвичай ємкість цих регістрів не перевищує ширину лінії даних (а найчастіше менше її), хоча деякі контроллери можуть використовувати як регістри чергу FIFO для буферизації інформації, що поступає.

Зрозуміло, набір регістрів і складових їх бітів приблизний, він покликаний послужити нам моделлю для опису процесу передачі інформації від обчислювальної системи до зовнішнього пристрою і назад, але в тому або іншому вигляді він зазвичай присутній у всіх контроллерах пристроїв.

Опит пристроїв і переривання. Виняткові ситуації і системні виклики

Побудувавши модель контроллера і уявляючи собі, що ховається за словами "прочитати інформацію з порту" і "записати інформацію в порт", ми готові до розгляду процесу взаємодії пристрою і процесора. Як і в попередніх випадках, прикладом нам послужить команда запису, тепер уже запису або виведення даних на зовнішній пристрій. У нашій моделі для виведення інформації, що поміщається в регістр вхідних даних, без перевірки успішності виводу процесор і контроллер повинні зв'язуватися таким чином.

1. Процесор в циклі читає інформацію з порту регістра станів і перевіряє значення біта зайнятості. Якщо біт зайнятості встановлений, то це означає, що пристрій ще не завершив попередню операцію, і процесор йде на нову ітерацію циклу. Якщо біт

зайнятості скинутий, то пристрій готовий до виконання нової операції, і процесор переходить на наступний крок.

2. Процесор записує код команди виводу в порт регістра управління.
3. Процесор записує дані в порт регістра вхідних даних.
4. Процесор встановлює біт готовності команди. У наступних кроках процесор не задіяний.
5. Коли контроллер помічає, що біт готовності команди встановлений, він встановлює біт зайнятості.
6. Контроллер аналізує код команди в регістрі управління і виявляє, що це команда виводу. Він бере дані з регістра вхідних даних і ініціює виконання команди.
7. Після завершення операції контроллер обнуляє біт готовності команди.
8. При успішному завершенні операції контроллер обнуляє біт помилки в регістрі стану, при невдалому завершенні команди – встановлює його.
9. Контроллер скидає біт зайнятості.

При необхідності виведення нової порції інформації всі ці кроки повторюються. Якщо процесор цікавить, коректно або некоректно була виведена інформація, то після кроку 4 він повинен в циклі прочитувати інформацію з порту регістра станів до тих пір, поки не буде скинутий біт зайнятості пристрою, після чого проаналізувати стан біта помилки.

Як видимий, на першому кроці (і, можливо, після кроку 4) процесор чекає звільнення пристрою, безперервно опитуючи значення біта зайнятості. Такий спосіб взаємодії процесора і контроллера отримав назву polling або, в російському перекладі, способу опиту пристроїв. Якщо швидкості роботи процесора і пристрою введення-виводу приблизно рівні, то це не приводить до істотного зменшення корисної роботи, що здійснюється процесором. Якщо ж швидкість роботи пристрою істотно менше швидкості процесора, то вказана техніка різко знижує продуктивність системи і необхідно застосовувати інший архітектурний підхід.

Для того, щоб процесор не чекав полягання готовності пристрою введення-виводу в циклі, а міг виконувати в цей час іншу роботу, необхідно, щоб пристрій само умів сигналізувати процесору про свою готовність. Технічний механізм, який дозволяє зовнішнім пристроям оповіщати процесор про завершення команди виводу або команди введення, отримав назву механізму переривань.

У простому випадку для реалізації механізму переривань необхідно до шин локальної магістралі, що є у нас, додати ще одну лінію, що сполучає процесор і пристрої введення-виводу – лінію переривань. Після закінчення виконання операції зовнішній пристрій виставляє на цю лінію спеціальний сигнал, по якому процесор після виконання чергової команди (або після завершення чергової ітерації при виконанні ланцюжкових команд, тобто команд, що повторюються циклічно із зрушенням по пам'яті) змінює свою поведінку. Замість виконання чергової команди з потоку команд він частково зберігає вміст своїх регістрів і переходить на виконання програми обробки переривання, розташованої за заздалегідь обумовленою адресою. За наявності тільки одній лінії переривань процесор при виконанні цієї програми повинен опитати стан всіх пристроїв введення-виводу, щоб визначити, від якого саме пристрої прийшло переривання (polling переривань!), виконати необхідні дії (наприклад, вивести в цей пристрій чергову порцію інформації або перевести відповідний процес із стану очікування в стан готовність) і повідомити пристрій, що переривання оброблене (зняти переривання).

У більшості сучасних комп'ютерів процесор прагнуть повністю звільнити від необхідності опиту зовнішніх пристроїв, у тому числі і від визначення за допомогою опиту пристрою, що згенерував сигнал переривання. Пристрої повідомляють про свою готовність процесор не

безпосередньо, а через спеціальний контроллер переривань, при цьому для спілкування з процесором він може використовувати не одну лінію, а цілу шину переривань. Кожному пристрою привласнюється свій номер переривання, який при виникненні переривання контроллер переривання заносить в свій регістр стани і, можливо, після розпізнавання процесором сигналу переривання і отримання від нього спеціального запиту виставляє на шину переривань або шину даних для читання процесором. Номер переривання зазвичай служить індексом в спеціальній таблиці переривань, що зберігається за адресою, що задається при ініціалізації обчислювальної системи, і що містить адреси програм обробки переривань – вектори переривань. Для розподілу пристроїв по номерах переривань необхідно, щоб від кожного пристрою до контроллера переривань йшла спеціальна лінія, відповідна одному номеру переривання. За наявності безлічі пристроїв таке підключення стає неможливим, і на один провідник (один номер переривання) підключається декілька пристроїв. В цьому випадку процесор при обробці переривання все одно вимушений займатися опитом пристроїв для визначення пристрою, що видав переривання, але в істотно меншому об'ємі. Зазвичай при установці в систему нового пристрою введення-виводу потрібний апаратний або програмно визначити, яким буде номер переривання, що виробляється цим пристроєм.

Розглядаючи кооперацію процесів і взаємовиключання, ми говорили про існування критичних секцій усередині ядра операційної системи, при виконанні яких необхідно виключити всякі переривання від зовнішніх пристроїв. Для заборони переривань, а точніше, для несприйнятності процесору до зовнішніх переривань зазвичай існують спеціальні команди, які можуть маскувати (забороняти) всі або деякі з переривань пристроїв введення-виводу. В той же час певні кризові ситуації в обчислювальній системі (наприклад, неусувний збій в роботі оперативної пам'яті) повинні вимагати її негайної реакції. Такі ситуації викликають переривання, які неможливо замаскувати або заборонити і які поступають в процесор по спеціальній лінії шини переривань, званою лінією немаскованих переривань (NMI – Non-Maskable Interrupt).

Не всі зовнішні пристрої є однаково важливими з погляду обчислювальної системи. Відповідно, деякі переривання є істотнішими, ніж інші. Контроллер переривань зазвичай дозволяє встановлювати пріоритети для переривань від зовнішніх пристроїв. При майже одночасному виникненні переривань від декількох пристроїв (під час виконання однієї і тієї ж команди процесора) процесору повідомляється номер найбільш пріоритетного переривання для його обслуговування насамперед. Менш пріоритетне переривання при цьому не пропадає, про нього процесору доповість після обробки пріоритетнішого переривання. Більш того, при обробці виниклого переривання процесор може отримати повідомлення про виникнення переривання з вищим пріоритетом і перемкнутися на його обробку.

Механізм обробки переривань, по якому процесор припиняє виконання команд в звичайному режимі і, частково зберігши свій стан, відволікається на виконання інших дій, опинився настільки зручний, що часто розробники процесорів використовують його і для інших цілей. Хоча ці випадки і не відносяться до операцій введення-виводу, ми вимушені згадати їх тут, для того, щоб їх не плутали з перериваннями. Схожим чином процесор обробляє виняткові ситуації і програмні переривання.

Для зовнішніх переривань характерні наступні особливості.

- Зовнішнє переривання виявляється процесором між виконанням команд (або між ітераціями у разі виконання ланцюжкових команд).
- Процесор при переході на обробку переривання зберігає частину свого стану перед виконанням наступної команди.

- Переривання відбуваються асихронно з роботою процесора і непередбачувано, програміст жодним чином не може передбачити, в якому саме місці роботи програми відбудеться переривання.

Виняткові ситуації виникають під час виконання процесором команди. До їх числа відносяться ситуації переповнювання, ділення на нуль, звернення до відсутньої сторінки пам'яті і так далі. Для виняткових ситуацій характерний наступне.

- Виняткові ситуації виявляються процесором під час виконання команд.
- Процесор при переході на виконання обробки виняткової ситуації зберігає частину свого стану перед виконанням поточної команди.
- Виняткові ситуації виникають синхронно з роботою процесора, але непередбачувано для програміста, якщо тільки той спеціально не змусив процесор ділити деяке число на нуль.

Програмні переривання виникають після виконання спеціальних команд, як правило, для виконання привілейованих дій усередині системних викликів. Програмні переривання мають наступні властивості.

- Програмне переривання відбувається в результаті виконання спеціальної команди.
- Процесор при виконанні програмного переривання зберігає свій стан перед виконанням наступної команди.
- Програмні переривання, природно, виникають синхронно з роботою процесора і абсолютно передбачені програмістом.

Треба сказати, що реалізація схожих механізмів обробки зовнішніх переривань, виняткових ситуацій і програмних переривань лежить цілком на совісті розробників процесорів. Існують обчислювальні системи, де всі три ситуації обробляються по-різному.

Прямий доступ до пам'яті (Direct Memory Access – DMA)

Використання механізму переривань дозволяє розумно завантажувати процесор в той час, коли пристрій введення-виводу займається своєю роботою. Проте запис або читання великої кількості інформації з адресного простору введення-виводу (наприклад, з диска) приводять до великої кількості операцій введення-виводу, які повинен виконувати процесор. Для звільнення процесора від операцій послідовного виведення даних з оперативної пам'яті або послідовного введення в неї був запропонований механізм прямого доступу зовнішніх пристроїв до пам'яті – ПДП або Direct Memory Access – DMA. Давайте коротко розглянемо, як працює цей механізм.

Для того, щоб який-небудь пристрій, окрім процесора, міг записати інформацію в пам'ять або прочитати її з пам'яті, необхідно щоб цей пристрій міг забрати у процесора управління локальною магістраллю для виставляння відповідних сигналів на шини адреси, даних і управління. Для централізації ці обов'язки зазвичай покладаються не на кожен пристрій окремо, а на спеціальний контроллер – контроллер прямого доступу до пам'яті. Контроллер прямого доступу до пам'яті має декілька спарених ліній – каналів DMA, які можуть підключатися до різних пристроїв. Перед початком використання прямого доступу до пам'яті цей контроллер необхідно запрограмувати, записавши в його порти інформацію про те, який канал або канали передбачається задіювати, які операції вони здійснюватимуть, яка адреса пам'яті є початковою для передачі інформації і яка кількість інформації має бути передане. Отримавши по одній з ліній – каналів DMA, сигнал запиту на передачу даних від зовнішнього пристрою, контроллер по шині управління повідомляє процесор про бажання узяти на себе управління локальною магістраллю. Процесор, можливо, через деякий час, необхідне для завершення його дій з магістраллю, передає управління нею контроллеру DMA, сповістившись його спеціальним

сигналом. Контроллер DMA виставляє на адресну шину адресу пам'яті для передачі черговій порції інформації і по другій лінії каналу прямого доступу до пам'яті повідомляє пристрій про готовність магістралі до передачі даних. Після цього, використовуючи шину даних і шину управління, контроллер DMA, пристрій введення-виводу і пам'ять здійснюють процес обміну інформацією. Потім контроллер прямого доступу до пам'яті сповіщає процесор про свою відмову від управління магістраллю, і той берет керівні функції на себе. При передачі великої кількості даних весь процес повторюється циклічно.

При прямому доступі до пам'яті процесор і контроллер DMA по черзі управляють локальною магістраллю. Це, звичайно, декілька знижує продуктивність процесора, оскільки при виконанні деяких команд або при читанні чергової порції команд у внутрішній кеш він повинен очікувати звільнень магістралі, але в цілому продуктивність обчислювальної системи істотно зростає.

При підключенні до системи нового пристрою, який уміє використовувати прямий доступ до пам'яті, зазвичай необхідно програмно або апаратний задати номер каналу DMA, до якого буде приписано пристрій. На відміну від переривань, де один номер переривання міг відповідати декільком пристроям, канали DMA завжди знаходяться в монопольному володінні пристроїв.

Логічні принципи організації введення-виводу

Розглянуті в попередньому розділі фізичні механізми взаємодії пристроїв введення-виводу з обчислювальною системою дозволяють зрозуміти, чому різноманітні зовнішні пристрої легко можуть бути додані в існуючі комп'ютери. Все, що необхідно зробити користувачеві при підключенні нового пристрою, – це відобразити порти пристрою у відповідний адресний простір, визначити, який номер відповідатиме перериванню, що генерується пристроєм, і, якщо потрібно, закріпити за пристроєм деякий канал DMA. Для нормального функціонування hardware це буде досить. Проте ми до цих пір нічого не сказали про те, як має бути побудована підсистема управління введенням-виводом в операційній системі для легені і безболісного додавання нових пристроїв і які функції взагалі зазвичай на неї покладаються.

Структура системи введення-виводу

Якщо доручити непідготовленому користувачеві сконструювати систему введення-виводу, здатну працювати зі всім безліччю зовнішніх пристроїв, то, швидше за все, він опиниться в ситуації, в якій знаходилися біологи і зоологи до появи праць Ліннея [Linnaes, 1789]. Всі пристрої різні, відрізняються по виконуваних функціях і своїх характеристиках, і здається, що принципово неможливо створити систему, яка без великих постійних переробок дозволяла б охоплювати все різноманіття видів. Ось перелік лише декількох напрямів (далеко не повний), по яких розрізняються пристрої.

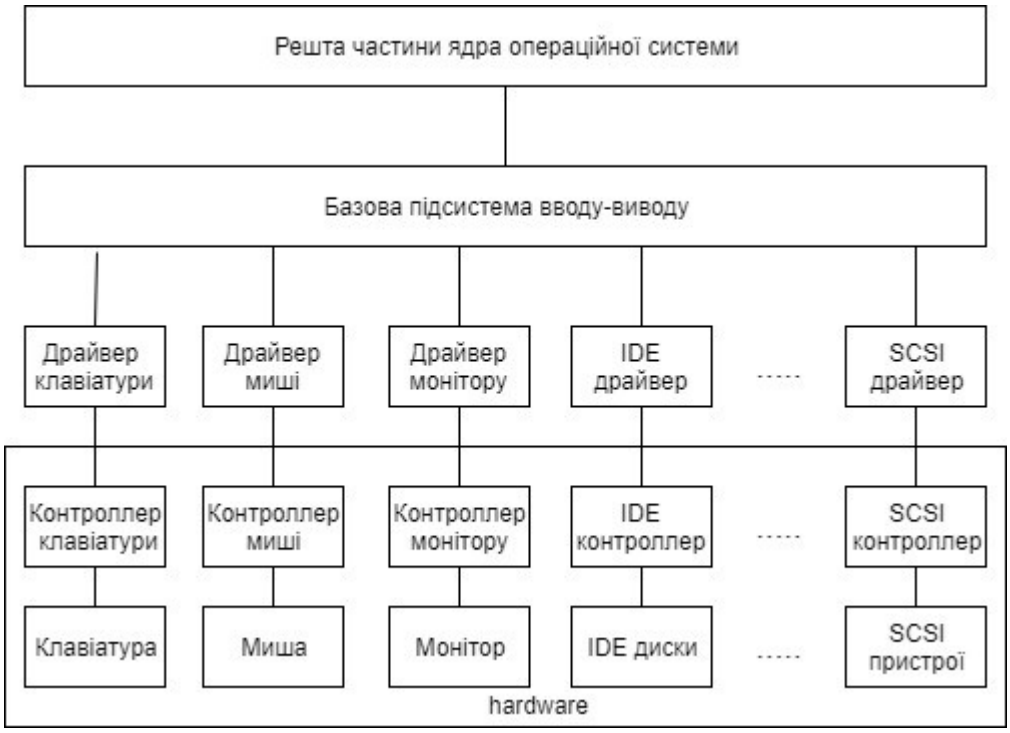
- Швидкість обміну інформацією може варіюватися в діапазоні від декількох байтів в секунду (клавіатура) до декількох гігабайтів в секунду (мережеві карти).
- Одні пристрої можуть використовуватися декількома процесами паралельно (є такими, що розділяються), тоді як інші вимагають монопольного захоплення процесом.
- Пристрої можуть запам'ятовувати виведену інформацію для її подальшого введення або не володіти цією функцією. Пристрої, що запам'ятовують інформацію, у свою чергу, можуть диференціюватися по формах доступу до збереженої інформації: забезпечувати до неї послідовний доступ в жорстко заданому порядку або уміти знаходити і передавати тільки необхідну порцію даних.
- Частина пристроїв уміє передавати дані тільки по одному байту послідовно (символьні пристрої), а частина пристроїв уміє передавати блок байтів як єдине ціле (блокові пристрої).
- Існують пристрої, призначені тільки для введення інформації, пристрої, призначені тільки для виведення інформації, і пристрої, які можуть виконувати і введення, і вивід.

В області технічного забезпечення вдалося виділити декілька основних принципів взаємодії зовнішніх пристроїв з обчислювальною системою, тобто створити єдиний інтерфейс для їх підключення, поклавши всі специфічні дії на контроллери самих пристроїв. Тим самим конструктори обчислювальних систем переклав весь клопіт, пов'язаний з підключенням зовнішньої апаратури, на розробників самої апаратури, примушуючи їх дотримуватися певного стандарту.

Схожий підхід виявився продуктивним і в області програмного підключення пристроїв введення-виводу. Подібно до того як Ліннею вдалося закласти основи систематизації знань про рослинний і тваринний світ, розділивши все живе в природі на відносно невелике число класів і загонів, ми можемо розділити пристрою на відносно невелике число типів, що відрізняються по набору операцій, які можуть бути ними виконані, вважаючи решту всіх відмінностей за неістотні. Ми можемо потім специфікувати інтерфейси між ядром операційної системи, що здійснює деяку

загальну політику введення-виводу, і програмними частинами, пристроями, що безпосередньо управляють, для кожного з таких типів. Більш того, розробники операційних систем дістають можливість звільнитися від написання і тестування цих специфічних програмних частин, що отримали назву драйверів, передавши цю діяльність виробникам самих зовнішніх пристроїв. Фактично ми приходимо до використання принципу рівневої або листкової побудови системи управління введенням-виводом для операційної системи (див. мал. 16.1).

Два нижні рівні цієї листкової системи складає hardware: самі пристрої, що безпосередньо виконують операції, і їх контроллери, службовці для організації спільної роботи пристроїв і решти обчислювальної системи. Наступний рівень складають драйвери пристроїв введення-виводу, що приховують від розробників операційних систем особливості функціонування конкретних приладів і забезпечують чітко певний інтерфейс між hardware і вищерозміщеним рівнем – рівнем базової підсистеми введення-виводу, яка, у свою чергу, надає механізм взаємодії між драйверами і програмною частиною обчислювальної системи в цілому.



Мал. 16.1. Структура системи введення-виводу

У подальших розділах ми докладніше розглянемо організацію і функції набору драйверів і базової підсистеми введення-виводу.

Систематизація зовнішніх пристроїв і інтерфейс між базовою підсистемою введення-виводу і драйверами

Як і система видів Ліннея, система типів пристроїв є далеко не повною і не строго витриманою. Пристрої зазвичай прийнято розділяти за переважаючим типом інтерфейсу на наступні види:

- символьні (клавіатура, модем, термінал і т. п.);
- блокові (магнітні і оптичні диски і стрічки, і т. д.);
- мережеві (мережеві карти);
- все решта (таймери, графічні дисплеї, телевізійні пристрої, відеокамери і т. п.);

Таке ділення є вельми умовним. У одних операційних системах мережеві пристрої можуть не виділятися в окрему групу, в деяких інших – окремі групи складають звукові пристрої і відеопристрої і так далі Деякі групи у свою чергу можуть розбиватися на підгрупи: підгрупа жорстких дисків, підгрупа мишок, підгрупа принтерів. Нас такі деталі не цікавлять. Ми не ставимо перед собою мету здійснити систематизацію всіх можливих пристроїв, які можуть бути підключені до обчислювальної системи. Єдине, для чого нам знадобиться ця класифікація, так це для ілюстрації того положення, що пристрої можуть бути розділені на групи по виконуваних ними функціях, і для розуміння функцій драйверів, і інтерфейсу між ними і базовою підсистемою введення-виводу.

Для цього ми розглянемо тільки дві групи пристроїв: символьні і блокові. Як вже згадувалося в попередньому розділі, символьні пристрої – це пристрої, які уміють передавати дані тільки послідовно, байт за байтом, а блокові пристрої – це пристрої, які можуть передавати блок байтів як єдине ціле.

До символьних пристроїв зазвичай відносяться пристрої введення інформації, які спонтанно генерують вхідні дані: клавіатура, миша, модем, джойстик. До них же відносяться і пристрої виведення інформації, для яких характерне представлення даних у вигляді лінійного потоку: принтери, звукові карти і так далі За своєю природою символьні пристрої зазвичай уміють здійснювати дві загальні операції: ввести символ (байт) і вивести символ (байт) – get і put.

Для блокових пристроїв, таких як магнітні і оптичні диски, стрічки і тому подібне природними є операції читання і запису блоку інформації – read і write, а також, для пристроїв прямого доступу, операція пошуку необхідного блоку інформації – seek.

Драйвери символьних і блокових пристроїв повинні надавати базовій підсистемі введення-виводу функції для здійснення описаних загальних операцій. Окрім загальних операцій, деякі пристрої можуть виконувати операції специфічні, властиві тільки ним – наприклад, звукові карти уміють збільшувати або зменшувати середню гучність звучання, дисплеї уміють змінювати свою роздільну здатність. Для виконання таких специфічних дій в інтерфейс між драйвером і базовою підсистемою введення-виводу зазвичай входить ще одна функція, що дозволяє безпосередньо передавати драйверу пристрою довільну команду з довільними параметрами, що дозволяє задіювати будь-яку можливість драйвера без зміни інтерфейсу. У операційній системі Unix така функція отримала назву ioctl (від input-output control).

Окрім функцій read, write, seek (для блокових пристроїв), get, put (для символьних пристроїв) і ioctl, до складу інтерфейсу зазвичай включають ще наступні функції.

- Функцію ініціалізації або повторної ініціалізації роботи драйвера і пристрою – open.
- Функцію тимчасового завершення роботи з пристроєм (може, наприклад, викликати відключення пристрою) – close.
- Функцію опиту стану пристрою (якщо по яких-небудь причинах робота з пристроєм проводиться методом опиту його стану, наприклад, в операційних системах Windows NT і Windows 9x така побудована робота з принтерами через паралельний порт) – poll.
- Функцію зупинки драйвера, яка викликається при зупинці операційної системи або вивантаженні драйвера з пам'яті, halt.

Існує ще ряд дій, виконання яких може бути покладене на драйвер, але оскільки, як правило, це дії базової підсистеми введення-виводу, ми поговоримо про них в наступному розділі. Приведені вище назви функцій, звичайно, є умовними і можуть мінятися від однієї

операційної системи до іншої, але дії, що виконуються драйверами, характерні для більшості операційних систем, і відповідні функції присутні в інтерфейсах до них.

Функції базової підсистеми введення-виводу

Базова підсистема введення-виводу служить посередником між процесами обчислювальної системи і набором драйверів. Системні виклики для виконання операцій введення-виводу трансформуються нею у виклики функцій необхідного драйвера пристрою. Проте обов'язки базової підсистеми не зводяться до виконання тільки дій трансляції загального системного виклику в звернення до приватної функції драйвера. Базова підсистема надає обчислювальній системі такі послуги, як підтримка системних викликів, що блокуються, не блокуються і асинхронних, буферизація і кешування вхідних і вихідних даних, здійснення spooling'а і монопольного захоплення зовнішніх пристроїв, обробка помилок і переривань, що виникають при операціях введення-виводу, планування послідовності запитів на виконання цих операцій. Давайте зупинимося на цих послугах докладніше.

Системні виклики, що блокуються, не блокуються і асинхронні

Всі системні виклики, пов'язані із здійсненням операцій введення-виводу, можна розбити на три групи по способах реалізації взаємодії процесу і пристрою введення-виводу.

- До першої, найбільш звичній для більшості програмістів групі відносяться системні виклики, що блокуються. Як випливає з самої назви, застосування такого виклику приводить до блокування процесу, що ініціював його, тобто процес переводиться операційною системою із стану виконання в стан очікування. Завершивши виконання всіх операцій введення-виводу, наказаних системним викликом, операційна система переводить процес із стану очікування в стан готовність. Після того, як процес буде знову вибраний для виконання, в ньому відбудеться остаточне повернення з системного виклику. Типовим для застосування такого системного виклику є випадок, коли процесу необхідно отримати від пристрою строго певну кількість даних, без яких він не може виконувати роботу далі.
- До другої групи відносяться системні виклики, що не блокуються. Їх назва не зовсім точно відображає суть справи. У простому випадку процес, що застосував виклик, що не блокується, не переводиться в стан очікування взагалі. Системний виклик повертається негайно, виконавши наказані йому операції введення-виводу повністю, частково або не виконавши зовсім, залежно від поточної ситуації (стани пристрою, наявність даних і т. д.). У складніших ситуаціях процес може блокуватися, але умовою його розблокування є завершення всіх необхідних операцій або закінчення деякого проміжку часу. Типовим випадком застосування системного виклику, що не блокується, може бути періодична перевірка на надходження інформації з клавіатури при виконанні трудомістких розрахунків.
- До третьої групи відносяться асинхронні системні виклики. Процес, що використав асинхронний системний виклик, ніколи в нім не блокується. Системний виклик ініціює виконання необхідних операцій введення-виводу і негайно повертається, після чого процес продовжує свою регулярну діяльність. Про закінчення завершення операції введення-виводу операційна система згодом інформує процес зміною значень деяких

змінних, передачею йому сигналу або повідомлення або яким-небудь іншим способом. Необхідно чітко розуміти різницю між викликами, що не блокуються і асинхронними. Системний виклик, що не блокується, для виконання операції read повернеться негайно, але може прочитати запитану кількість байтів, менша кількість або взагалі нічого. Асинхронний системний виклик для цієї операції також повернеться негайно, але необхідна кількість байтів рано чи пізно буде прочитана в повному об'ємі.

Буферизація і кешування

Під буфером зазвичай розуміється деяка область пам'яті для запам'ятовування інформації при обміні даних між двома пристроями, двома процесами або процесом і пристроєм. Обмін інформацією між двома процесами відноситься до області кооперації процесів, і ми детально розглянули його організацію у відповідній лекції. Тут нас цікавитиме використання буферів у тому випадку, коли одним з учасників обміну є зовнішній пристрій. Існує три причини, що приводять до використання буферів в базовій підсистемі введення-виводу.

- Перша причина буферизації – це різні швидкості прийому і передачі інформації, якими володіють учасники обміну. Розглянемо, наприклад, випадок передачі потоку даних від клавіатури до модему. Швидкість, з якою поставляє інформацію клавіатура, визначається швидкістю набору тексту людиною і звичайна істотно менше швидкості передачі даних модемом. Для того, щоб не позичати модем на весь час набору тексту, роблячи його недоступним для інших процесів і пристроїв, доцільно накопичувати введену інформацію в буфері або декількох буферах достатнього розміру і посилати її через модем після заповнення буферів.
- Друга причина буферизації – це різні об'єми даних, які можуть бути прийняті або отримані учасниками обміну одноразово. Візьмемо інший приклад. Хай інформація поставляється модемом і записується на жорсткий диск. Окрім володіння різними швидкостями здійснення операцій, модемом і жорстким диском є пристрої різного типу. Модем є символьним пристроєм і видає дані байт за байтом, тоді як диск є блоковим пристроєм і для проведення операції запису для нього потрібно накопичити необхідний блок даних в буфері. Тут також можна застосовувати більш за один буфер. Після заповнення першого буфера модем починає заповнювати другою, одночасно із записом першого на жорсткий диск. Оскільки швидкість роботи жорсткого диска в тисячі разів більша, ніж швидкість роботи модему, до моменту заповнення другого буфера операція запису першого буде завершена, і модем знову зможе заповнювати перший буфер одночасно із записом другого на диск.
- Третя причина буферизації пов'язана з необхідністю копіювання інформації із застосувань, що здійснюють уведення-виведення, в буфер ядра операційної системи і назад. Допустимо, що деякий призначений для користувача процес побажав вивести інформацію зі свого адресного простору на зовнішній пристрій. Для цього він повинен виконати системний виклик з узагальненою назвою write, передавши як параметри адресу області пам'яті, де розташовані дані, і їх об'єм. Якщо зовнішній пристрій тимчасово зайнятий, то можлива ситуація, коли до моменту його звільнення вмісту

потрібної області виявиться зіпсованим (наприклад, при використанні асинхронної форми системного виклику). Щоб уникнути виникнення подібних ситуацій, найпростіше на початку роботи системного виклику скопіювати необхідні дані в буфер ядра операційної системи, що постійно знаходиться в оперативній пам'яті, і виводити їх на пристрій з цього буфера.

Під словом кеш (cash – "готівка"), етимологію якого ми тут не розглядатимемо, зазвичай розуміють область швидкої пам'яті, що містить копію даних, розташованих де-небудь в повільнішій пам'яті, призначену для прискорення роботи обчислювальної системи. Ми з вами стикалися з цим поняттям при розгляді ієрархії пам'яті. У базовій підсистемі введення-виводу не слід змішувати два поняття, буферизацію і кешування, хоча часто для виконання цих функцій відводиться одна і та ж область пам'яті. Буфер часто містить єдиний набір даних, що існує в системі, тоді як кеш за визначенням містить копію даних, що існують де-небудь ще. Наприклад, буфер, використовуваний базовою підсистемою для копіювання даних з призначеного для користувача простору процесу при виводі на диск, може у свою чергу застосовуватися як кеш для цих даних, якщо операції модифікації і повторного читання даного блоку виконуються достатньо часто.

Функції буферизації і кешування не обов'язково мають бути локалізовані в базовій підсистемі введення-виводу. Вони можуть бути частково реалізовані в драйверах і навіть в контроллерах пристроїв, скритно по відношенню до базової підсистеми.

Spooling і захоплення пристроїв

Про поняття spooling ми говорили в першій лекції нашого курсу, як про механізм, що вперше дозволив сумістити реальні операції введення-виводу одного завдання з виконанням іншого завдання. Тепер ми можемо визначити це поняття точніше. Під словом spool ми маємо на увазі буфер, що містить вхідні або вихідні дані для пристрою, на якому слід уникати чергування його використання різними процесами. Правда, в сучасних обчислювальних системах spool для введення даних практично не використовується, а в основному призначений для накопичення вихідної інформації.

Розглянемо як зовнішній пристрій принтер. Хоча принтер не може друкувати інформацію, що поступає одночасно від декількох процесів, може виявитися бажаним дозволити процесам здійснювати вивід на принтер паралельно. Для цього операційна система замість передачі інформації безпосередньо на принтер накопичує дані, що виводяться, в буферах на диску, організованих у вигляді окремого spool-файла для кожного процесу. Після завершення деякого процесу відповідний йому spool-файл ставиться в чергу для реального друку. Механізм, що забезпечує подібні дії, і отримав назву spooling.

У деяких операційних системах замість використання spooling для усунення race condition застосовується механізм монопольного захоплення пристроїв процесами. Якщо пристрій вільний, то один з процесів може отримати його в монопольне розпорядження. При цьому всі інші процеси при спробі здійснення операцій над цим пристроєм будуть або блоковані (переведені в стан очікування), або отримають інформацію про неможливість виконання операції до тих пір, поки процес, що захопив пристрій, не завершиться або явно не повідомить операційну систему про свою відмову від його використання.

Забезпечення spooling і механізму захоплення пристроїв є прерогативою базової підсистеми введення-виводу.

Обробка переривань і помилок

Якщо при роботі із зовнішнім пристроєм обчислювальна система не користується методом опитування його стану, а задіює механізм переривань, то при виникненні переривання, як ми вже говорили раніше, процесор, частково зберігши свій стан, передає управління спеціальній програмі обробки переривання. Ми вже розглядали дії операційної системи над процесами, що відбуваються при виникненні переривання, в розділі "Перемикання контексту" лекції 2, де після виникнення переривання здійснювалися наступні дії: збереження контексту, обробка переривання, планування використання процесора, відновлення контексту. Тоді ми звертали більше уваги на дії, пов'язані із збереженням і відновленням контексту і плануванням використання процесора. Тепер давайте докладніше зупинимося на тому, що ховається за словами "обробка переривання".

Одна і та ж процедура обробки переривання може застосовуватися для декількох пристроїв введення-виводу (наприклад, якщо ці пристрої використовують одну лінію переривань, що йде від них до контроллера переривань), тому перша дія власне програми обробки полягає у визначенні того, яке саме пристрій видало переривання. Знаючи пристрій, ми можемо виявити процес, який ініціював виконання відповідної операції. Оскільки переривання виникає як при вдалому, так і при невдалому її виконанні, наступне, що ми повинні зробити, – це визначити успішність завершення операції, перевіривши значення біта помилки в регістрі стану пристрою. В деяких випадках операційна система може зробити певні дії, направлені на компенсацію виниклої помилки. Наприклад, у разі виникнення помилки читання з гнучкого диска можна спробувати кілька разів повторити виконання команди. Якщо компенсація помилки неможлива, то операційна система згодом сповістить про це процес, що запитав виконання операції (наприклад, спеціальним кодом повернення з системного виклику). Якщо цей процес був заблокований до виконання операції, що завершилася, то операційна система переводить його в стан готовність. За наявності інших незадоволених запитів до пристрою, що звільнився, операційна система може ініціювати виконання наступного запиту, одночасно сповістившись пристрій, що переривання оброблене. На цьому, власне, обробка переривання закінчується, і система може приступати до планування використання процесора.

Дії з обробки переривання і компенсації виникаючих помилок можуть бути частково перекладені на плечі відповідного драйвера. Для цього до складу інтерфейсу між драйвером і базовою підсистемою введення-виводу додають ще одну функцію – функцію обробки переривання intr.

Планування запитів

При використанні системного виклику, що не блокується, може опинитися, що потрібний пристрій вже зайнятий виконанням деяких операцій. Виклик, що в цьому випадку не блокується, може негайно повернутися, не виконавши запитаних команд. При організації запиту на здійснення операцій введення-виводу за допомогою виклику, що блокується або асинхронного, зайнятість пристрою приводить до необхідності постановки запиту в чергу до даного пристрою. В результаті з кожним пристроєм виявляється зв'язаний список незадоволених запитів процесів, що знаходяться в стані очікування, і запитів, що виконуються в асинхронному режимі. Стан очікування розщеплюється на набір черг процесів, що чекають різних пристроїв введення-виводу (або чекаючих зміни станів різних об'єктів – семафорів, черг повідомлень, умовних змінних в моніторах і так далі – див. лекцію 6).

Після завершення виконання поточного запиту операційна система (по ходу обробки виниклого переривання) повинна вирішити, який із запитів в списку має бути задоволений наступним, і ініціювати його виконання. Точно так, як і для вибору чергового процесу на виконання із списку готових нам доводилося здійснювати короткострокове планування процесів, тут нам необхідно здійснювати планування застосування пристроїв, користуючись

яким-небудь алгоритмом цього планування. Критерії і цілі такого планування мало відрізняються від критеріїв і цілей планування процесів.

Завдання планування використання пристрою зазвичай покладається на базову підсистему введення-виводу, проте для деяких пристроїв кращі алгоритми планування можуть бути тісно пов'язані з деталями їх внутрішнього функціонування. У таких випадках операція планування переноситься всередину драйвера відповідного пристрою, оскільки ці деталі приховані від базової підсистеми. Для цього в інтерфейс драйвера додається ще одна спеціальна функція, яка здійснює вибір чергового запиту, – функція strategy.

Висновок

Функціонування будь-якої обчислювальної системи зазвичай зводиться до виконання двох видів роботи: обробка інформації і операції по здійсненню її введення-виводу. З погляду операційної системи "обробкою інформації" є тільки операції, що здійснюються процесором над даними, що знаходяться в пам'яті на рівні ієрархії не нижче чим оперативна пам'ять. Все останнє відноситься до "операцій введення-виводу", тобто до обміну інформацією із зовнішніми пристроями.

Не дивлячись на все різноманіття пристроїв введення-виводу, управління їх роботою і обмін інформацією з ними будуються на відносно невеликій кількості принципів. Основними фізичними принципами побудови системи введення-виводу є наступні: можливість використання різних адресних просторів для пам'яті і пристроїв введення-виводу; підключення пристроїв до системи через порти введення-виводу, що відображаються в один з адресних просторів; існування механізму переривання для сповіщення процесора про завершення операцій введення-виводу; наявність механізму прямого доступу пристроїв до пам'яті, минувши процесор.

Механізм, подібний до механізму переривань, може використовуватися також і для обробки виключень і програмних переривань, проте це цілком лежить на совісті розробників обчислювальних систем.

Для побудови програмної частини системи введення-виводу характерний "листковий" підхід. Для безпосередньої взаємодії з hardware використовуються драйвери пристроїв, що приховують від решти частини операційної системи всі особливості їх функціонування. Драйвери пристроїв через жорстко певний інтерфейс пов'язані з базовою підсистемою введення-виводу, в обов'язки якої входять: організація роботи системних викликів, що блокуються, не блокуються і асинхронних, буферизація і кешування вхідних і вихідних даних, здійснення spooling і монопольного захоплення зовнішніх пристроїв, обробка помилок і переривань, що виникають при операціях введення-виводу, планування послідовності запитів на виконання цих операцій. Доступ до базової підсистеми введення-виводу здійснюється за допомогою системних викликів.

Частина функцій базової підсистеми може бути делегована драйверам пристроїв і самим пристроям введення-виводу.