ЗМІСТ

1. Вступ....5
   1.1. Предмет і задачі курсу....5
   1.2. Рекомендації з літератури....5
   1.3. Короткий опис історії ОС....6
   1.4. Класифікація ОС....10
   1.5. Критерії оцінювання ОС....11
   1.6. Основні функції і структура ОС....14
   1.7. ОС, що використовуються в подальшому викладі....14
2. Керування пристроями....14
   2.1. Основні задачі управління пристроями....14
   2.2. Класифікація периферійних пристроїв і їхня архітектура....14
   2.3. Переривання....14
   2.4. Архітектура підсистеми введення/виводу....14
   2.5. Способи організації введення/виводу....14
   2.6. Буферизація і кешування....14
   2.7. Драйвери пристроїв....14
   2.8. Керування пристроями в MS-DOS....14
   2.9. Управління пристроями в Windows....14
   2.10. Керування пристроями в UNIX....14
3. Управління даними....14
   3.1. Основні задачі керування даними....14
   3.2. Характеристики файлів і архітектура файлових систем....14
   3.3. Розміщення файлів....14
   3.4. Захист даних....14
   3.5. Поділ файлів між процесами....14
   3.6. Файлова система FAT і управлення даними в MS-DOS....14
   3.7. Файлові системи і керування даними в UNIX....14
   3.8. Файлова система NTFS і управлення даними в Windows 8....14
4. КЕРУВАННЯ ПРОЦЕСАМИ....14
   4.1. Основні задачі управління процесами....14
   4.2. Реалізація багатозадачного режиму....14
   4.3. Проблеми взаємодії процесів....14
   4.4. Керування процесами в MS-DOS....14
   4.5. Управління процесами в Windows....14
   4.6. Керування процесами в UNIX....14
5. Управління пам'яттю....14
   5.1. Основні задачі керування пам'яттю....14
   5.2. Віртуальні і фізичні адреси....14

5.3. Розподіл пам'яті без використання віртуальних адрес....14
5.4. Сегментна організація пам'яті....14
5.5. Сторінкова організація пам'яті....14
5.6. Порівняння сегментної і сторінкової організації....14
5.7. Керування пам'яттю в MS-DOS....14
5.8. Управління пам'яттю в Windows....14
5.9. Керування пам'яттю в UNIX....14
6. Література....14

1. ВСТУП

1.1. Предмет і задачі курсу

Предметом вивчення в даному курсі є ***операційні системи*** (ОС) сучасних комп'ютерів.

У першому наближенні ОС можна визначити як комплекс програм, що забезпечують інтерфейс між апаратурою комп'ютера, прикладними програмами і користувачем комп'ютера. Відповідно до цього визначенню, усі функції, виконувані ОС, підлеглі рішенню двох основних задач:

0* організації ефективної роботи апаратури комп'ютера;

1* забезпеченню зручного використання ресурсів комп'ютера як прикладними програмами, так і користувачем, що працює з комп'ютером.

Основною метою курсу є вивчення пристрою і функціонування сучасних ОС. При цьому будуть розглядатися два кола питань:

2* основні принципи побудови ОС, найбільш розповсюджені алгоритми виконання різних функцій ОС, типові структури даних, використовувані для забезпечення роботи ОС;

3* практичне втілення цих принципів, алгоритмів, структур у найбільш розповсюджених сучасних ОС.

У задачі курсу не входить навчання практичним прийомам роботи з конкретними ОС. Це набагато краще робити самостійно. З іншого боку, не ставиться і задача навчити слухачів розробляти нові ОС. Операційні системи не є масовими виробами, і брати участь у їхній розробці доводиться лише меншої частини програмістів. Рівень знань, якого хотілося б досягти при вивченні даного предмета, можна порівняти з тим рівнем знань про пристрій автомобіля, що корисний гарному водієві. Він не обов'язково повинний бути автомеханіком, однак повинний в основних рисах розуміти, що знаходиться під капотом і як воно там крутиться.

1.2. Рекомендації з літератури

Зміст лекційного курсу не обов'язково на 100% збіжиться з даним конспектом, тому надійніше за все ходити на лекції і мати до іспиту власний конспект.

З книг загального характеру підручник /1/ найбільше відповідає даному курсові, як по змісту, так і в ще більшому ступені по загальному погляді на предмет. Цю книгу можна знайти й у Мережі, а також у локальній мережі кафедри.

Похвали і поваги заслуговує книга /2/ - величезний по обсязі і досить простій по викладу огляд усього важливого про ОС.

Досить гарна також книга /3/, використовувана як основний підручник по ОС у багатьох американських університетах.

Ледарям придасться книга /4/, у якій, поряд з іншими питаннями системного програмування, коротко і досить толково викладені основні проблеми ОС. Правда, книга старувата.

Книги /5/ і /6/ містять багато корисного по практичних питаннях проектування ОС, а /7/ залишається гарним джерелом по теоретичних і алгоритмічних питаннях.

З літератури по Windows випливає насамперед рекомендувати класичну книгу /8/, що робить зрозумілими багато питань, важко перетравлювані по офіційній документації. Більш глибокий розбір того, «як це зроблено в Windows», можна знайти в книзі /9/. На жаль, ця книга помітно уступає чудової, але застарілої по матеріалі книзі того ж автора /10/,яку, проте, корисно прочитати тим, кого цікавлять питання практичної реалізації ОС.

На тлі незлічимих і неотличимых друг від друга користувальницьких посібників з UNIX варто виділити досить серйозну роботу /11/.Не втратила інтересу старенька, тонка книжка /12/,у якій утримується багато корисного про основні структури даних і алгоритмах UNIX. Набагато докладніше ті ж питання розглянуті в іншій старій книзі, що давно придбала популярність в електронному варіанті /13/.Паперове видання цієї книги російською мовою існує тільки в піратському варіанті, без вказівки імені автора.

Для тих, кого ще цікавить MS-DOS, можна порекомендувати /14/,це одна з кращих книг на дану тему.

Деякі алгоритми, використовувані при реалізації різних ОС, добре викладені в класичній книзі /15/.

Великі колекції літератури і документації по ОС маються в Інтернету. Серед російськомовних сайтов можна рекомендувати, наприклад, /16,17,18,19,20/.

Знання англійської мови відкриває доступ до моря свіжої інформації в Інтернету. Величезна купа зведень по Windows утримується в /21/.На сайте /22/можна знайти цікаві статті по окремих питаннях архітектури Windows, а також скачати ряд корисних утиліт. З великого числа сайтов, присвячених UNIX і Linux, можна назвати, наприклад, /23/і /24/.На сайте /25/можна знайти багато статей і книг по актуальних питаннях програмування, у тому числі по ОС.

Основна література

1. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.
2. Silberschatz A. Operating System Concepts. – 7th ed. / A.Silberschatz, P.B.Galvin, G.Gagne. – John Wiley & Sons Inc. – 2005. – 944 p. (у електронному варіанті).
3. Харт Дж.М. Систкмное программирование в среде Windows, 3-е изд.: Пер. с англ. – М.: Изд. дом «Вильямс», 2005. – 592 с.
4. Microsoft Developer Network Library 2010 (доступна через інтернет та локально на цифрових носіях).
5. Unix man (доступна через інтернет та локально на цифрових носіях).
6. Лав Роберт. Linux. Системное программирование - 1-е издание. – СПб: “Питер”, 2008. – 416 с.
7. Джонсон Харт. Системное программирование в среде Windows - 3-е издание. – М.: “Вильямс”, 2005. – 592 с.
8. Зубков С.В. Assembler для DOS, Windows и UNIX. – Изд. 2-е испр. и доп. – М. : ДМК Пресс, 2000. – 608 с.

9. Брябрин В.М. Программное обеспечение персональних ЭВМ. – М. : Наука. 2002.
10. Данилочкин В.П. и др., Операционная система ОС ЕС. – М.: Финансы и статистика, 2004.

Додаткова література

1. Prof. John Kubiatowicz, відділ комп'ютерних наук, Університет штату Каліфорнія (Берклі). [Електронний ресурс]. – Режим доступу: http://www.cs.berkeley.edu/~kubitron/courses/cs162-F09/
2. Проф. Ален Дікс, кафедра комп'ютингу, Хаддерсфілдський Університет (Велика Британія). [Електронний ресурс]. – Режим доступу: http://www.hiraeth.com/alan/tutorials/courses/unixprog.html

Dr. Santanu Chaudhury, Кафедра електронної інженерії, IIT Delhi (Індія). [Електронний ресурс]. – Режим доступу: http://www.youtube.com/view_play_list?p=5677C301A37CEF76

1.3. Короткий нарис історії ОС

Вивчення історії розвитку ОС показує, що всі істотні просування в області архітектури ОС зв'язані з впливом двох основних факторів:

4* прогрес технології, що приводить до швидкого зростання характеристик апаратури ЕОМ і до появи принципово нових типів апаратури;

5* принципово нові ідеї, що виникають у проектувальників.

Не вплутуючи в давню суперечку матеріалістів з ідеалістами, у даному окремому випадку приходиться визнати, що перший, матеріальний фактор визначав розвиток ОС на 80 – 90%. Такі технологічні прориви, як винахід магнітних дисків, мікропроцесорів, створення високоякісних відеомоніторів, настійно вимагали радикальних змін у технології роботи з комп'ютером, і внаслідок цього обумовлювали створення принципове нових типів ОС або їхніх окремих підсистем. З іншого боку, деякі ідеї в області організації обчислювального процесу й інтерфейсу дали серйозний поштовх удосконалюванню архітектури комп'ютерів.

Не знаючи хоча б загалом основних етапів розвитку апаратного і програмного забезпечення, важко зрозуміти багато особливостей сучасних ОС.

Додатковий аргумент на користь знання історії полягає в тім, що багато технічних рішень, що, здавалося, назавжди пішли в минуле разом з конкретними системами, зненацька знову виявляються актуальними на новому витку розвитку. Деякі приклади такого роду будуть розглянуті в курсі.

1.3.1. Передісторія ОС

Незабаром після того, як наприкінці 40-х років XX століття були створені перші електронні комп'ютери, дуже гостро встала проблема підвищення ефективності використання устаткування, і насамперед центрального процесора.

Типовий комп'ютер першого – другого поколінь являв собою велику кімнату, заставлену шафами і повиту кабелями. Кожне з основних пристроїв – центральний процесор, оперативна пам'ять, нагромаджувачі на магнітних стрічках, пристрою введення з перфокарт, принтер –

займало один або трохи «шаф» або «тумб», наповнених радіолампами і механічними частинами.

Усе це коштувало великих грошей, споживало скажену кількість електроенергії і регулярно ламалося.

У таких умовах машинний час коштувало дуже дорого. Проте, звичайна практика використання ЕОМ не сприяла економії. Як правило, програміст, що розробляє програму, замовляв щодня кілька годин машинного часу і протягом цього часу монопольно використовував машину. Виконавши черговий запуск отлаживаемой програми (которую треба було щораз вводити або з клавіатури, або, у кращому випадку, з перфокарт), користувач одержував роздруківку (найчастіше у виді масиву цифр), аналізував результати, вносив зміни в програму і знову запускав неї. Таким чином, у ході сеансу налагодження дороге устаткування простоювало 99% часу, поки програміст осмислював результати і працював із пристроями введення/висновку. Крім того, збій при введенні однієї перфокарти міг зажадати почати спочатку всю роботу програми.

Виникла велика ідея – використовувати сам комп'ютер для підвищення ефективності роботи з ним же.

Одне з відгалужень цієї ідеї – створення мов і систем програмування – розглядається в окремих курсах. Іншим важливим кроком стало покладання на спеціальну комп'ютерну програму частини тих функцій, що до цього виконував оператор або сам програміст.

Програми такого роду називалися звичайно ***моніторами*** (не плутати з монітором як пристроєм висновку, що у той час був найрідшою екзотикою!). Монітор приймав команди, що складаються, як правило, з 1-2 букв назви і 1-3 аргументів, заданих 8-ричными або 16-ричными числами. Типовими командами були, наприклад:

6* завантаження даних з перфокарт по зазначеній адресі пам'яті;

7* перегляд і коректування (із друкарської машинки) значень у зазначеному діапазоні адрес;

8* покрокове виконання програми з видачею результатів кожної команди на друкарську машинку;

9* запуск програми з зазначеної адреси з завданням адрес контрольних крапок зупинки.

Незважаючи на убогість, по нинішніх мірках, подібних засобів, вони у свій час значно підвищили продуктивність роботи програмістів. Однак кардинального підвищення завантаження процесора не відбулося.

Часом широкого поширення моніторів у світі були 50-і роки минулого століття (у СРСР – 60-і роки). В даний час щось подібне можна зустріти на самих примітивних мікропроцесорних контролерах.

1.3.2. Пакетні ОС

Історію власне ОС можна почати з появи наприкінці 50-х років перших систем, що організують роботу з пакетного принципу.

Найважливішою організаційною зміною, що происшли на цьому етапі розвитку, стало масове вигнання програмістів з машинних залів, як фактора, що лише вносить сум'яття в роботу.

Тепер від програміста було потрібно зібрати пакет перфокарт, що містить його програму, дані до неї, а також керуючі перфокарти. Ці карти на спеціально розробленій ***мові керування завданнями*** (JCL, Job Control Language) пояснювали операційній системі, чиє це завдання, що потрібно зробити з програмою (наприклад, передати її трансляторові з Фортрану), що почати у випадку успішної трансляції (імовірно, пустити на рішення), що – при наявності помилок (наприклад, перейти до іншої програми), відкіля узяти вихідні дані (наприклад, з таке-те циліндра магнітного диска). Крім того, там могли бути навіть указівки на те, скільки метрів

папера можна виділити на роздруківку і який максимальний час може зайняти робота програми.

Обійтися без настільки докладних інструкцій було не можна, тому що програміст не був присутній при запуску завдання і не міг втрутитися особисто.

Підготовлений пакет передавався, разом з іншими подібними пакетами, операторові ЕОМ, перед яким стояли дві основні задачі: щоб у пристрої введення не переводилися пакети завдань і щоб у принтері не скінчився папір. Коли процесор закінчував обробку завдання і печатка його результатів, він уводив наступний пакет і приступав до його обробки. Так досягалася основна мета пакетного режиму – виключити простої процесора через нерозторопність людей.

Незабаром розроблювачі ОС усвідомили, що вичерпано далеко не всі резерви підвищення завантаження процесора. Операції введення і печатки вимагали лише дуже невеликої частки від повної продуктивності процесора. Крім того, у ході роботи програми траплялися звертання до периферійних пристроїв (наприклад, до магнітних стрічок і, пізніше, дискам), при виконанні яких процесор знову простоював. Доцільно було знайти спосіб, щоб у ці періоди чекання завантажити процесор іншою роботою. Але для цього необхідно, щоб у пам'яті процесора знаходилися відразу кілька програм, тоді ОС змогла б переключати процесор на виконання тієї програми, що у даний момент може працювати.

Така організація роботи, коли в пам'яті знаходяться кілька програм і система у визначені моменти переключає виконання з однієї програми на іншу, була названа ***мультипрограммированием***. Ця важлива ідея в різних утіленнях пережила ті пакетні системи, у яких вона вперше була реалізована, і є основою для функціонування практично всіх сучасних ОС.

Серед найбільш розвитих пакетних ОС з мультипрограммированием не можна не назвати OS/360, основну ОС знаменитого в 60-70 р. сімейства ЕОМ IBM 360/370.

1.3.3. ОС з поділом часу

На рубежі 60-70 р. розповсюдженим і не занадто дорогим периферійним пристроєм стають монітори (спочатку монохромні і працюючі тільки в текстовому режимі). При цьому процесор і ОЗУ залишаються найдорожчими і громіздкими пристроями обчислювальної системи. У цих умовах виникає і швидко здобуває популярність принципово новий тип ОС – ***системи з поділом часу***.

До одній ЕОМ підключається кілька десятків робочих місць, обладнаних дисплеєм (монітор + клавіатура) і спільно використовує обчислювальні ресурси ЕОМ. Процесорний час поділяється на кванти тривалістю в кілька десятків миллисекунд і після закінчення кожного кванта процесор може бути переключений на обслуговування іншого процесу, іншого дисплея. Оскільки тепер підготовку текстів програм виконують самі програмісти за дисплеями, а робота з редагування тексту вимагає дуже малих витрат процесорного часу, процесор встигає обслужити всі робочі місця практично без відчутної затримки. Велика частина часу процесора приділяється невеликому числу робочих місць, де в даний момент запущені на виконання програми. При цьому, зрозуміло, середня швидкість роботи кожної програми зменшується, принаймні в стільки разів, скільки програм виконується одночасно.

Режим поділу часу став величезним полегшенням для програмістів, що знову змогли до деякої міри відчути себе «хазяїнами» ЕОМ і одержали можливість запускати програми на трансляцію і налагодження хоч кожні 5 хвилин. Це дозволило скоротити терміни розробки і налагодження програм.

Для трудомістких обчислювальних завдань, що передбачають рахунок по раніше налагоджених програмах, режим поділу часу менш ефективний, чим пакетний, оскільки часте переключення процесора між виконуваними програмами вимагає додаткових витрат часу.

Системи поділу часу використовуються в режимі діалогу з користувачем, тому замість громіздких, деталізованих операторів JCL у них використовуються більш прості команди, що виконують елементарні дії – запуск програми, видача на екран файлу або каталогу, копіювання або видалення файлу і т.п. Користувачеві не потрібно передбачати заздалегідь усі можливі исходы виконання команди, набагато простіше побачити результат виконання на екрані і після цього прийняти рішення, яку команду виконувати наступної. У той же час, деякі часто повторювані послідовності команд зручно описати один раз у виді «пакетного завдання» і потім використовувати при необхідності. У цьому плані системи поділу часу зберігають ті зручні можливості, що надавали пакетні системи.

Спочатку як апаратну основу систем поділу часу повинні були використовуватися «великі» ЕОМ, що пізніше стало прийняте називати «мейнфреймами» (mainframes). Пізніше, у міру прогресу обчислювальної техніки, це стало по плечу навіть миниэвм (так називався в ті роки клас комп'ютерів, що займали усього лише один-два невеликих шафки). Варто особливо згадати серію миниэвм PDP-11, що мала найширше поширення в усім світі протягом півтора десятків років.

Цей період (70-і роки у світі, 80-і в СРСР) характерний глибоким розвитком теорії і практики створення могутніх ОС, що містять розвиті засоби керування процесами і пам'яттю, що реалізують многопользовательский режим роботи. З великого числа подібних систем особливого згадування заслуговує UNIX – єдина система, що благополучно дожила до нашого часу.

1.3.4. Однозадачные ОС для ПЭВМ

У середині 70-х років був винайдений мікропроцесор, а до початку 80-х мікропроцесори стали доганяти по функціональних характеристиках раніше використовувалися «великі» процесори. Ця ситуація зробила майже марним режим поділу часу: навіщо поділяти один процесор між багатьма задачами і багатьма користувачами, якщо простіше і дешевше дати окремий мікропроцесор кожному користувачеві? Поділ часу залишилося доцільним хіба що у відношенні суперкомп'ютерів.

Поява і бурхливе поширення персональних комп'ютерів (ПК) викликало до життя нове покоління ОС, що виявилися в багато разів простіше своїх попередниць. Непотрібної виявився многопользовательская захист. Спочатку показалася непотрібної і многозадачность. Усе це можна було розцінити як явний регрес у розвитку ОС.

Найбільш популярної ОС для ранніх восьмиразрядных ПК була система CP/M відомої тоді фірми Digital Research, однак з появою на початку 80-х знаменитої машини IBM PC лідерство була міцно перехоплена системою MS-DOS фірми Microsoft.

1.3.5. Багатозадачні ОС для ПК із графічним інтерфейсом

Швидкий розвиток технології привело до того, що до кінця 80-х років ПК виявилися в стані вирішувати значно більш складні і трудомісткі задачі, чим раніш. При цьому багато хто з досягнень колишніх етапів розвитку ОС виявилися знову затребуваними, але тепер вже в нових умовах, серед яких треба назвати різке підвищення потужності процесорів і обсягу пам'яті, поява високоякісних графічних моніторів і розвиток мережних технологій.

Стала реальної така річ, як многозадачная ОС для ПК. Треба сказати, що спочатку ідея системи, у якій один користувач запускає одночасно кілька додатків, більшості фахівців здавалася порожнім піжонством і викликала глузування: «Чому б не виконати кілька програм по черзі?». Зараз з таким поглядом смішно навіть сперечатися.

А все-таки, як би ви обґрунтували користь многозадачности для сучасних ОС типу Windows?

На зміну ОС, що виконували текстові команди, що вводяться користувачем із клавіатури,

прийшли системи, у яких взаємодія з користувачем засновано на використанні GUI (Graphical User Interface, графічний інтерфейс користувача).

Значна частина ПК працює в складі локальних обчислювальних мереж. Це привело до того, що питання захисту даних користувача знову придбали першорядне значення.

1.4. Класифікація ОС

Існують різні види класифікації ОС по тим або інших ознаках, що відбивають різні істотні характеристики систем.

10* По призначенню.

11*Системи загального призначення. Це досить розпливчаста назва має на увазі ОС, призначені для рішення широкого кола задач, включаючи запуск різних додатків, розробку і налагодження програм, роботу з мережею і з мультимедиа.

12* ***Системи реального часу***. Цей важливий клас систем призначений для роботи в контурі керування об'єктами (такими, як літальні апарати, технологічні установки, автомобілі, складна побутова техніка і т.п.). З подібного призначення випливають тверді вимоги до надійності й ефективності системи. Повинне бути забезпечене точне планування дій системи в часі (керуючі сигнали повинні видаватися в задані моменти часу, а не просто «по можливості швидко»). Особливий підклас складають системи, ***убудовані*** в устаткування. Такі системи роками можуть виконувати фіксований набір програм, не вимагаючи втручання людини-оператора на більш глибокому рівні, чим натискання кнопки «Вкл.».

Іноді виділяють також такий клас ОС, як системи з «нежорстким» реальним часом. Це такі системи, що не можуть гарантувати точне дотримання тимчасових співвідношень, але «дуже намагаються», тобто містять засобу для пріоритетного виконання завдань, критичних за часом. Такій системі не можна довірити керування ракетою, але вона цілком справиться з демонстрацією відеофільму. Виділення подібних систем в окремий клас має скоріше рекламне значення, дозволяючи таким системам, як Windows NT і деякі версії UNIX, теж називати себе «системами реального часу».

13* Інші спеціалізовані системи. Це різні ОС, орієнтовані насамперед на ефективне рішення задач визначеного класу, з великим або меншим збитком для інших задач. Можна виділити, наприклад, мережні системи (такі, як Novell Netware), що забезпечують надійне і високоефективне функціонування локальних мереж.

14* По характері взаємодії з користувачем.

15* Пакетні ОС, що обробляють заздалегідь підготовлені завдання.

16* Діалогові ОС, що виконують команди користувача в інтерактивному режимі. Красиве слово «інтерактивний» означає постійна взаємодія системи з користувачем.

17* ОС із графічним інтерфейсом. У принципі, їх також можна віднести до діалогових систем, однак використання миші і всього, що з нею зв'язане (меню, кнопки і т.п.) вносить свою специфіку.

18* Убудовані ОС, не взаємодіючі з користувачем.

19* По числу одночасно виконуваних задач.

20* Однозадачные ОС. У таких системах у кожен момент часу може існувати не більш ніж один активний користувальницький процес. Варто помітити, що одночасно з ним можуть працювати системні процеси (наприклад, що виконують запити на введення/висновок).

21* Многозадачные ОС. Вони забезпечують рівнобіжне виконання декількох користувальницьких процесів. Реалізація многозадачности вимагає значного ускладнення алгоритмів і структур даних, використовуваних у системі.

22* По числу користувачів.

23* Однокористувальницькі ОС. Для них характерний повний доступ користувача до ресурсів системи. Подібні системи прийнятні в основному для ізольованих комп'ютерів, що не допускають доступу до ресурсів даного комп'ютера по мережі або з вилучених терміналів.

24* Многопользовательские ОС. Їхнім важливим компонентом є засоби захисту даних і процесів кожного користувача, засновані на понятті власника ресурсу і на точній указівці прав доступу, наданих кожному користувачеві системи.

25* По апаратурній основі.

26* Однопроцесорні ОС. У даному курсі будуть розглядатися тільки вони.

27* Многопроцессорные ОС. У задачи такої системи входить, крім іншого, ефективне розподіл виконуваних завдань по процесорах і організація погодженої роботи всіх процесорів.

28* Мережні ОС. Вони включають можливість доступу до інших комп'ютерів локальної мережі, роботи з файловими й іншими серверами.

29* Розподілені ОС. Їхня відмінність від мережних полягає в тім, що розподілена система, використовуючи ресурси локальної мережі, представляє їх користувачеві як єдину систему, не розділену на окремі машини.

1.5. Критерії оцінки ОС

При порівняльному розгляді різних ОС у цілому або їхніх окремих підсистемах виникає вічне питання – яка з них краще і чому, яка архітектура системи переважніше, який з алгоритмів ефективніше, яка структура даних зручніше і т.п.

Дуже рідко можна дати однозначна відповідь на подібні питання, якщо мова йде про практично використовувані системи. Система або її частина, що гірше інших систем у всіх відносинах, просто не мала би права на існування. Насправді має місце типова многокритериальная задача: мається кілька важливих критеріїв якості, і система, що випереджає інші по одному критерії, звичайно уступає по іншому. Порівняльна важливість критеріїв залежить від призначення системи й умов її роботи.

1.5.1. Надійність

Цей критерій узагалі прийнятий вважати найважливішим при оцінці програмного забезпечення, і у відношенні ОС його дійсно беруть до уваги в першу чергу.

Що розуміється під надійністю ОС?

Насамперед, її ***живучість***, тобто здатність зберігати хоча б мінімальну працездатність в умовах апаратних збоїв і програмних помилок. Висока живучість особливо важлива для ОС комп'ютерів, убудованих в апаратуру, коли втручання людини утруднене, а відмовлення комп'ютерної системи може мати важкі наслідки.

По-друге, здатність, як мінімум, діагностувати, а як максимум, компенсувати хоча б деякі типи апаратних збоїв. Для цього звичайно вводиться надмірність збереження найбільш важливі дані системи.

По-третє, ОС не повинна містити власних (внутрішніх) помилок. Це вимога рідка буває

здійсненно в повному обсязі (програмісти давно зуміли довести своїм замовникам, що в будь-якій великій програмі завжди є помилки, і це в порядку речей), однак випливає хоча б домогтися, щоб основні, часто використовувані або найбільш відповідальні частини ОС були вільні від помилок.

Нарешті, до надійності системи варто віднести її здатність протидіяти явно нерозумним діям користувача. Звичайний користувач повинний мати доступ тільки до тих можливостям системи, що необхідні для його роботи. Якщо ж користувач, навіть діючи в рамках своїх повноважень, намагається зробити щось дуже дивне (наприклад, отформатировать системний диск), то найменше, що повиннао зробити ОС, це перепитати користувача, чи упевнений він у правильності своїх дій.

1.5.2. Ефективність

Як відомо, ефективність будь-якої програми визначається двома групами показників, які можна узагальнено назвати «час» і «пам'ять». При розробці системи приходиться приймати багато непростих рішень, зв'язаних з оптимальним балансом цих показників.

Найважливішим показником временнуй ефективності є ***продуктивність*** системи, тобто усереднена кількість корисної обчислювальної роботи, виконуваної в одиницю часу. З іншого боку, для діалогових ОС не менш важливо ***час реакції*** системи на дії користувача. Ці показники можуть до деякої міри суперечити один одному. Наприклад, у системах поділу часу збільшення кванта часу підвищує продуктивність (за рахунок скорочення числа переключень процесів), але погіршує час реакції.

У програмуванні відома аксіома: виграш у часі досягається за рахунок програшу в пам'яті, і навпаки. Це повною мірою відноситься до ОС, розроблювачам яких постійно приходиться шукати баланс між витратами часу і пам'яті.

Турбота від ефективності довгий час стояла не першому місці при розробці програмного забезпечення, і особливо ОС. На жаль, зворотним боком стрімкого збільшення потужності комп'ютерів стало ослаблення інтересу до ефективності програм. В даний час ефективність є першорядною вимогою хіба що у відношенні систем реального часу.

1.5.3. Зручність

Цей критерій найбільш суб'єктивний. Можна запропонувати, наприклад, такий підхід: система або її частина зручна, якщо вона дозволяє легко і просто вирішувати ті задачі, що зустрічаються найбільше часто, але в той же час містить засобу для рішення широкого кола менш стандартних задач (нехай навіть ці засоби не настільки прості). Приклад: така часта дія, як копіювання файлу, повинне виконуватися за допомогою однієї простої команди або легкого руху миші; у той же час для зміни розділів диска не гріх почитати керівництво, оскільки це може знадобитися навіть не щороку.

Розроблювачі кожної ОС мають власні представлення про зручність, і кожна ОС має своїх прихильників, що вважають саме її ідеалом зручності.

1.5.4. Масштабируемость

Досить дивний термін «масштабируемость» (scalability) означає можливість настроювання системи для використання в різних варіантах, у залежності від потужності обчислювальної системи, від набору конкретних периферійних пристроїв, від ролі, що грає конкретний комп'ютер (сервер, робоча станція або ізольований комп'ютер) від призначення комп'ютера (домашній, офісний, дослідницький і т.п.).

Гарантією масштабируемости служить продумана модульна структура системи, що дозволяє в ході установки системи збирати і набудовувати потрібну конфігурацію. Можливий і

інший підхід, коли під загальною назвою поєднуються, по суті, різні системи, що забезпечують у розумних межах програмну сумісність. Прикладом можуть служити версії Windows NT/2000/XP, Windows 95/98 і Windows CE.

У деяких випадках фірми, що роблять програмне забезпечення, штучно відключають у більш дешевих версіях системи ті можливості, що насправді реалізовані, але стають доступні, тільки якщо користувач купує ліцензію на більш дорогу версію. Але це вже питання, зв'язаний не з технічною стороною справи, а з маркетинговою політикою.

1.5.5. Здатність до розвитку

Щоб складна програма мала шанси проіснувати довго, у неї споконвічно повинні бути закладені можливості для майбутнього розвитку.

Однією з головних умов здатності системи до розвитку є добре продумана модульна структура, у якій чітко визначені функції кожного модуля і його взаємозв'язку з іншими модулями. При цьому створюється можливість удосконалювання окремих модулів з мінімальним ризиком викликати небажані наслідки для інших частин системи.

Важливою вимогою до розвитку ОС є ***сумісність версій знизу нагору***, що означає можливість безболісного переходу від старої версії до нового, без утрати раніше напрацьованих прикладних програм і без необхідності різкої зміни всіх навичок користувача. Зворотна сумісність – зверху вниз – як правило, не гарантується, оскільки в ході розвитку система здобуває нові можливості, не реалізовані в старих версіях. Програма з Windows 3.1 буде нормально працювати й у Windows XP; навпаки – навряд чи.

Фірми-виробники ОС додають максимум зусиль для забезпечення сумісності знизу нагору, щоб не віджахнути користувачів. Але при цьому фірми намагаються в кожну нову версію закласти який-небудь новий цукерок, що спонукала би користувачів якомога швидше купити неї.

Сумісність версій – благо для користувача, однак на практиці вона часто приводить до консервації давно віджилих свій вік особливостей або ж просто невдалих рішень, прийнятих у ранній версії системи. У документації подібні архаїзми позначаються як «застарілі» (obsolete), але повного відмовлення від них, як правило, не відбувається (а раптом десь ще працює прикладна програма, написана двадцять років тому з використанням саме цих засобів?).

Як правило, найбільш консервативною стороною будь-який ОС є не алгоритми, а структури системних даних, тому далекоглядні розроблювачі заздалегідь будують структури «на виріст»: закладають у них резервні поля, використовують перемінні замість деяких констант, установлюють кількісні обмеження з великим запасом і т.п.

1.5.6. Мобільність

Під ***мобільністю*** (portability) розуміється можливість переносу програми (у даному випадку ОС) на іншу апаратну платформу, тобто на інший тип процесора й іншу архітектуру комп'ютера. Тут мається на увазі перенос з помірними трудозатратами, не потребуючої повної переробки системи.

Властивість мобільності не настільки однозначно позитивно, як може показатися. Щоб програма була мобільна, при її розробці варто відмовитися від глибокого використання особливостей конкретної архітектури (таких, як кількість і функціональні можливості регістрів процесора, нестандартні команди і т.п.). Мобільна програма повинна бути написана мовою досить високого рівня (часто використовується мова С), якому можна реалізувати на комп'ютерах будь-якої архітектури. Платою за мобільність завжди є деяка втрата ефективності, тому немобільні системи поширені досить широко.

З іншого боку, історія системного програмування засіяна останками чудовими, ефективними і зручних, але немобільних ОС, що вимерли разом із процесорами, для яких вони

призначалися. У той же час мобільна система UNIX продовжує процвітати четвертий десяток років, набагато переживши ті комп'ютери, для яких вона спочатку створювалася. Приблизно 5-10% вихідних текстів UNIX написані на мови ассемблера і повинні листуватися заново при переносі на нову архітектуру. Інша частина системи написана на С і практично не вимагає змін при переносі.

Деяким компромісом є ***многоплатформенные*** ОС (наприклад, Windows NT), споконвічно спроектовані для використання на декількох апаратних платформах, але не гарантуюча можливість переносу на нові, не передбачені заздалегідь архітектури.

1.6. Основні функції і структура ОС

Відповідно до багаторічної традиції, при розгляді основ функціонування ОС прийнято виділяти чотири основних групи функцій, виконуваних системою.

30* ***Керування пристроями***. Маються на увазі всі периферійні пристрої, що підключаються до комп'ютера, – клавіатура, монітор, принтери, диски і т.п.

31* ***Керування даними***. Під цим стародавнім терміном зараз розуміється робота з файлами, хоча були часи, коли звертання до даних на магнітних носіях виконувалося шляхом вказівки адреси розміщення даних на пристрої, а поняття файлу не існувало.

32* ***Керування процесами***. Ця сторона роботи ОС зв'язана з запуском і завершенням роботи програм, обробкою помилок, забезпеченням рівнобіжної роботи декількох програм на одному комп'ютері.

33* ***Керування пам'яттю***. Оперативна пам'ять комп'ютера – це такий ресурс, якого завжди не вистачає. У цих умовах розумне планування використання пам'яті є найважливішим чинником ефективної роботи.

Мається ще кілька важливих обов'язків, що лягають на ОС, що важко втиснути в рамки традиційної класифікації функцій. До них, насамперед, відносяться наступні.

34* ***Організація інтерфейсу з користувачем***. Форми інтерфейсу можуть бути різноманітними, у залежності від типу і призначення ОС: мова керування пакетами завдань, набір діалогових команд, засобу графічного інтерфейсу.

35* ***Захист даних***. Як тільки система перестає бути надбанням одного ізольованого від зовнішнього світу користувача, питання захисту даних від несанкціонованого доступу здобувають першорядну важливість. ОС, що забезпечує роботу в мережі або в системі поділу часу, повинна відповідати наявним стандартам безпеки.

36* ***Ведення статистики***. У ході роботи ОС повинна збиратися, зберігатися й аналізуватися різноманітна інформація: про кількість часу, витраченій різними програмами і користувачами, про інтенсивність використання ресурсів, про спроби некоректних дій користувачів, про збої устаткування і т.п. Зібрана інформація зберігається в системних журналах і в облікових записах користувачів.

Для розуміння роботи ОС необхідно уміти виділяти основні частини системи і їхнього зв'язку, тобто описувати структуру системи. Для різних ОС їхній структурний розподіл може бути досить різним. Найбільш загальними видами структуризації можна вважати два. З одного боку, можна вважати, що ОС розділено на підсистеми, що відповідають перерахованим вище групам функцій. Такий розподіл достатньо обґрунтовано, програмні модулі ОС дійсно в основному можна віднести до однієї з цих підсистем. Інший важливий структурний розподіл зв'язаний з поняттям ***ядра*** системи.

Ядро, як можна зрозуміти з назви, це основна, «сама системна» частина операційної системи. Маються різні визначення ядра. Відповідно до одному з них, ядро – це ***резидентная***

частина системи, тобто до ядра відноситься той програмний код, що постійно знаходиться в пам'яті протягом усієї роботи системи. Інші модулі ОС є ***транзитними***, тобто довантажуються в пам'ять з диска в міру необхідності на час своєї роботи. До транзитних частин системи відносяться:

37* ***утиліти*** (utilities) – окремі системні програми, що вирішують приватні задачі, такі як форматування і перевірку диска, пошук даних у файлах, моніторинг (відстеження) роботи системи і багато чого іншого;

38* ***системні бібліотеки підпрограм***, що дозволяють прикладним програмам використовувати різні спеціальні можливості, підтримувані системою (наприклад, бібліотеки для графічного висновку, для роботи з мультимедиа і т.п.);

39* ***інтерпретатор команд*** – програма, що виконує введення команд користувача, їхній аналіз і виклик інших модулів для виконання команд;

40* ***системний завантажник*** – програма, що при запуску ОС (наприклад, при включенні харчування) забезпечує завантаження системи з диска, її ініціалізацію і старт;

41* інші види програм, у залежності від конкретної системи.

Не менш важливим є визначення ядра, засноване на розрізненні режимів роботи комп'ютера. Усі сучасні процесори підтримують, як мінімум, два режими: ***привілейований*** режим (він же режим ядра, kernel mode) і ***непривілейований*** (режим задачі, режим користувача, user mode). Програми, що працюють у режимі ядра, мають повний, необмежений доступ до всіх ресурсів комп'ютера: його командам, адресам, портам уведення/висновку і т.п. У режимі задачі можливості програми обмежені, вона, зокрема, не може виконати деякі спеціальні команди. Апаратне розмежування можливостей є абсолютно необхідною умовою реалізації надійного захисту даних у многопользовательской системі. Звідси випливає і визначення ядра як частини ОС, що працює в режимі ядра. Всі інші програми, як системні утиліти, так і програми користувачів, працюють у режимі користувача і повинні звертатися до ядра для виконання багатьох системних дій.

Варто сказати, що переходи з режиму користувача в режим ядра і назад – це дії, що вимагають визначеного часу, і занадто часте їхнє виконання може привести до помітного зниження швидкості роботи програм. У зв'язку з цим визначення того, які функції повинні підтримуватися ядром, а які краще виконувати в режимі користувача – це непроста і важлива задача, що повинні вирішити розроблювачі ОС.

Особливу роль у структурі системи грають ***драйвери пристроїв***. Ці програми, призначені для обслуговування конкретних периферійних пристроїв, безсумнівно, можна віднести до ядра системи: вони майже завжди є резидентными і працюють у режимі ядра. Але на відміну від самого ядра, що змінюється тільки з появою нової версії ОС, набір використовуваних драйверів досить мобільний і залежить від набору пристроїв, підключених до даного комп'ютера. У деяких системах (наприклад, у ранніх версіях UNIX) для підключення нового драйвера було потрібно перекомпілювати все ядро. У більшості сучасних ОС драйверів підключаються до ядра в процесі завантаження системи, а іноді дозволяється навіть завантаження і вивантаження драйверів у ході роботи системи.

Як програмний інтерфейс системи, тобто засобів для звертання прикладних програм до послуг ОС, використовується документований набір ***системних викликів*** або ***функцій API*** (Applied Programming Interface). Між цими двома термінами є деяка різниця. Під системними викликами розуміються функції, реалізовані безпосередньо програмами ядра системи. При їхньому виконанні відбувається перехід з режиму користувача в режим ядра, а потім назад. На відміну від цього, API-функції визначаються як функції, описані в документації ОС, незалежно від того, чи виконуються вони ядром або ж системними бібліотеками, що працюють у режимі

користувача. У Windows часто кілька різних API-функцій звертаються до тому самому недокументированному системного виклику, але мають різні частини, що обрамляють, працюючі в режимі користувача.

Там, де розходження між двома цими поняттями несуттєво, можна використовувати нейтральний термін «***системні функції***».

1.7. ОС, використовувані надалі викладі

У наступних розділах курсу будуть розглядатися основні функції ОС і способи їхньої реалізації. Виклад загальних підходів буде доповнюватися прикладами, що відносяться головним чином до трьох широко відомих ОС:

42* MS-DOS – приклад простий однозадачной системи;

43* Windows – складна сучасна система, що виросла на базі MS-DOS;

44* UNIX – система, по можливостях порівнянна з Windows, однак разюче відрізняє по наборі основних концепцій і методам реалізації.

1.7.1. MS-DOS

Система MS-DOS була розроблена в 1981 р. спеціально для тільки що що з'явилася першої 16-розрядної ПЭВМ IBM PC на базі процесора i86. Перша версія системи була жахлива, але працездатна. В наступні роки фірмі Microsoft удалося значно поліпшити свою систему, хоча деякі пережитки першої версії виявилися невигубні. Альянс із фірмою IBM дозволив Microsoft домогтися фантастичного фінансового успіху.

MS-DOS являє собою однозадачную, однокористувальницьку, діалогову ОС. Вона веде діалог з користувачем у текстовому режимі й у більшому ступені розрахований на обслуговування прикладних програм текстового режиму, хоча допускає і графікові. Робота з мишею повинна забезпечуватися самими прикладними програмами при мінімальній підтримці з боку ОС. Для розміщення програми користувача і для своїх власних нестатків MS-DOS дозволяє використовувати 640 Кбайт пам'яті, що здавалося величезною завбільшки ті незапам'ятні часи акуратного програмування і повної відсутності файлів AVI і MP3. Пізніше були додані засоби, що дозволяють з деяким зусиллям використовувати до 4 Мб пам'яті.

Інтерфейс MS-DOS із прикладними програмами заснований на викликах програмних переривань, оброблюваних системою. Бульшую частина цих переривань прийнято називати ***функціями DOS***.

Система MS-DOS з'явилася стартовою площадкою для створення Windows. В даний час MS-DOS тихо відмирає, хоча усі версії Windows намагаються забезпечити виконання більшої частини програм, розроблених для їхньої попередниці.

У даному курсі MS-DOS розглядається як найбільш життєвий приклад простий і добре вивченої однозадачной системи для порівняння з більш могутніми многозадачными системами.

1.7.2. Windows

Система Windows була спочатку розроблена фірмою Microsoft як графічна оболонка, що завантажується поверх MS-DOS. Ідеї GUI (Graphic User Interface – графічний інтерфейс користувача) були вперше розроблені для експериментальної машини Xerox PARC ще в 70-х рр., потім підхоплені в MacOS – операційній системі комп'ютера Macintosh, відкіля і були з деякими погіршеннями запозичені в Windows. Версію Windows 1.0, що вийшла в 1985 р. і працювала на 1 Мб пам'яті з вікнами, що неперекриваються, прийнято розглядати як цікаву іграшку. Версія 2.0 (1987 р.) була більш серйозна, а версії 3.0 і 3.1 (1990-1992 р.), призначені для процесорів i386 і

використовують до 16 Мб пам'яті, уже мали великий успіх.

Усі перераховані версії продовжували залишатися надбудовами над MS-DOS, що використовують наявну файлову систему, але добавляющие своє власне керування процесами, пам'яттю і пристроями. За рахунок цього комбінацію DOS + Windows можна було назвати многозадачной однокористувальницької ОС із графічним інтерфейсом користувача.

У 1993 р. Microsoft випустила Windows NT – повноцінну многозадачную і многопользовательскую ОС, уже не засновану на MS-DOS. Однак, оскільки NT висувала підвищені вимоги до потужності процесора й обсягові пам'яті, у 1995 р. була випущена компромісна система Windows 95, що призначалася для заміни Windows 3.x у масового користувача. Підвищення швидкості роботи з порівняння з версією NT було досягнуто ціною відмовлення від многопользовательской захисти й ослаблення надійності системи. У Windows 95 неакуратно написана прикладна програма може привести до краху системи, а в Windows NT система краще ізольована від програм користувача. У той же час, практично всі коректно написані програми можуть переноситися з Windows 95 у Windows NT і навпаки.

Якийсь час дві лінії Windows розвивалися паралельно. Чергові версії Windows NT одержали назва Windows 2000, Windows XP, Windows 2003. Лінія Windows 95 була продовжена непринципово відрізняються від неї версіями Windows 98 і Windows ME, але далі, видимо, розвиватися не буде. Microsoft вважає, що сучасний рівень продуктивності ПЭВМ знімає необхідність у полегшеній версії системи.

Windows надає в розпорядження прикладних програм кілька тисяч документованих API-функцій на усі випадки життя.

Сучасна Windows – досить могутня і надто складна система, що має безліч достоїнств і недоліків, що неможливо обговорити коротко. Відзначимо, що широкому поширенню Windows, крім особливого положення фірми Microsoft на ринку, сприяє простота установки системи, що дозволяє рядовому користувачеві обійтися без допомоги фахівців.

Надалі викладі опис можливостей Windows буде в основному орієнтовано на лінію Windows NT/2000/XP.

1.7.3. UNIX

ОС UNIX була спочатку розроблена в 1969 р. співробітниками фірми Bell Laboratories Кеном Томпсоном і Деннисом Ритчи. У 1971 р. система була перенесена на машини надзвичайно розповсюдженої в 70-і роки серії PDP-11, а в 1973 р. Ритчи переписав систему мовою С, залишивши лише мінімум тексту мовою ассемблера. У перше десятиліття існування UNIX і сама система, і її вихідні тексти поширювалися вільно, що привело до надзвичайної популярності системи в наукових колах і університетах. Удосконалення системи могли вноситися кожним бажаючої й обговорювалися «усім миром». Зворотним боком такої відкритості стали труднощі стандартизації UNIX. Однак у 1988-1990 р. був розроблений набір стандартів, що одержав назву POSIX (Portable OS, а закінчення IX – як натяк на UNIX). Ці стандарти фіксували сучасні вимоги до систем типу UNIX з обліком теоретичних і практичних досягнень за минулі роки.

Починаючи з перших версій, UNIX являє собою многозадачную, многопользовательскую систему поділу часу. Основними достоїнствами UNIX є її висока мобільність, добре продуманий програмний і користувальницький інтерфейс. Загальновизнаною особливістю UNIX є внутрішня краса, елегантність основних архітектурних рішень. Як відомо, краса програми є вірною ознакою її вдалої конструкції і дає підстави сподіватися, що програма здатна до удосконалювання і буде служити довго. Багаторічна історія UNIX підтверджує це правило.

До недоліків UNIX можна віднести більш низьку ефективність і надійність роботи, що

значною мірою є платою за мобільність. Традиційна модель безпеки UNIX не відповідає сучасним вимогам, тому в різні комерційні версії приходиться включати додаткові засоби захисту даних. Широкому поширенню UNIX заважає також те, що процедури установки і настроювання системи не так прості, як у Windows, і при їхньому виконанні бажана участь програміста.

У 80-і роки були спроби перетворити UNIX у комерційну систему. Однак у 1991-1994 р. Линус Торвальдс, у той час студент-програміст із Хельсінкі, заново написав систему, що відповідає стандартам POSIX, але відрізняється від традиційної UNIX більшою надійністю й ефективністю. Ця система одержала назву Linux. Вихідні тексти Linux вільно поширюються, що дозволяє, як у часи молодості UNIX, розвивати систему загальними зусиллями величезного співтовариства зацікавлених програмістів. Ефективної координації цих зусиль дуже сприяє Інтернет. Трохи пізніше був відкритий вільний доступ до текстів відомої версії UNIX FreeBSD.

Архітектура UNIX, спочатку призначена для систем поділу часу з одним процесором, згодом виявилася цілком підходящої для підтримки мережних систем. Значна частина серверів Інтернету працює під керуванням тієї або іншої версії UNIX.

В даний час відбувається відчутне сближение різних типів ОС, призначених для підтримки тих самих типів обчислювальних систем. Сучасні версії UNIX і Windows надають досить близькі функціональні можливості, хоча найчастіше в зовсім різній формі. Вирівнюються також характеристики надійності і продуктивності систем.

Істотною відмінністю UNIX від Windows залишається місце, займане в системі засобами графічного інтерфейсу. Якщо в Windows вікна й усе, що з ними зв'язане, є невід'ємною частиною архітектури системи, то для UNIX за традицією основним засобом інтерфейсу з користувачем є текстова консоль. Ті або інші засоби віконного інтерфейсу, звичайно, присутні в сучасних UNIX-системах, але як додаткова, необов'язкова надбудова скоріше прикладного, чим системного характеру.

Дуже цікавою особливістю UNIX є розвита мова команд **`shell`**, що дозволяє не тільки вести елементарний діалог із системою, але і писати своєрідні програми (скрипты), за допомогою яких часто удається вирішити необхідну задачу, не прибігаючи до розробки нової програми на одній із традиційних мов програмування.

2. КЕРУВАННЯ ПРИСТРОЯМИ

2.1. Основні задачі керування пристроями

Поняття периферійних пристроїв (ПУ) поєднує, по суті, всі основні апаратурні блоки комп'ютера, за винятком процесора й основної пам'яті. Характерними рисами сучасних обчислювальних систем є широка розмаїтість типів і моделей ПУ, а також швидкий прогрес технологій, що приводить до постійного збільшення продуктивності пристроїв і до появи додаткових можливостей апаратури.

Найважливішими задачами будь-який ОС є забезпечення надійної роботи ПУ, ефективне використання всіх можливостей пристроїв.

Для підвищення надійності можуть використовуватися різні програмний^-програмні-апаратно-програмні методи, такі як дублювання даних, використання помехозащищенных кодів, контрольних сум даних і т.п. Процедури виконання операцій із пристроями повинні забезпечувати, як мінімум, виявлення апаратних помилок і збоїв, а як максимум – їхню компенсацію за рахунок надмірності даних і повторного виконання операцій.

Ефективність використання пристроїв означає насамперед скорочення часу, затрачуваного на обмін даними. Крім підвищення швидкості обміну, скорочення часу може досягатися за рахунок распараллеливания роботи ПУ і процесора. Могутнім фактором підвищення продуктивності системи є скорочення кількості операцій уведення/висновку за рахунок збереження даних у пам'яті для наступного використання.

Велика розмаїтість використовуваних пристроїв і постійна поява нових моделей диктують необхідність такої структури системи, що дозволяла б легке подключение нових пристроїв. Широке поширення одержує технологія «Plug & Play», тобто можливість оперативного приєднання пристроїв без вимикання комп'ютера.

При всій розмаїтості ОС повинна забезпечувати максимально можливу стандартизацію роботи з ними, щоб зміни апаратури не приводили до необхідності постійно модифікувати прикладне програмне забезпечення.

До числа додаткових задач, розв'язуваних підсистемами керування пристроями сучасних ОС, можна віднести збереження даних у стиснутому виді, шифрування даних і т.п.

2.2. Класифікація периферійних пристроїв і їхня архітектура

Під ***програмною архітектурою*** (або просто – ***архітектурою***) пристрою ми будемо розуміти сукупність тих структурних особливостей, що впливають на роботу програм із пристроєм. Наприклад, форма рознімання для підключення пристрою не входить у його архітектуру, але кількість і призначення ліній у цьому рознімanні може в нее входити (якщо ці лінії можуть програмно керуватися).

Як правило, разом із пристроєм поставляється його ***контролер*** (адаптер), що містить електронні схеми керування пристроєм. Конструктивно контролер може являти собою плату, що вставляється в рознімання шини комп'ютера, або може бути розташований у корпусі пристрою. У будь-якому випадку програми працюють із пристроєм за посередництвом його контролера, а тому з погляду архітектури немає розходження між поняттями «пристрій» і «контролер пристрою».

Класифікація периферійних пристроїв може бути виконана по різних ознаках.

45* Пристрою ***послідовного доступу*** (sequential access) і пристрою ***довільного доступу*** (random access). Для послідовних пристроїв характерна наявність визначеного природного порядку даних, при цьому обробка даних в іншому порядку або неможлива, або вкрай утруднена. Класичним прикладом є магнітні стрічки, для яких читання і запис даних ведуться від початку стрічки до кінця, а спроба доступу в іншому порядку зажадає постійного перемотування стрічки, що різко знижує швидкість роботи. До пристроїв послідовного доступу можна віднести також клавіатуру, мишу, принтер, модем.

Для пристроїв довільного доступу можливе звертання до різних порцій даних у будь-якому порядку, причому ефективність роботи не залежить (або слабко залежить) від порядку звертання. Для таких пристроїв характерна наявність адресації даних і операції пошуку потрібної адреси. Найбільш відомий приклад – магнітні диски й інші дискові пристрої. Крім того, до пристроїв довільного доступу можна віднести монітор ПК (там є адресація точек-пикселов, хоча операція пошуку не потрібна).

46* ***Символьні (байтовые***) і ***блокові*** пристрої. Для символьних пристроїв найменшою порцією що вводяться і виведених даних є один байт. Для деяких символьних пристроїв можна за одну операцію виконати введення або висновок кожного (у розумних межах) необхідної кількості байт.

Для блокових пристроїв найменшою порцією введення/висновку, виконуваного за одне

звертання до пристрою, є один блок, рівний, як правило, 2^k байт. Типовим розміром блоку може бути 512 байт, 1K байт, 4K байт і т.п., у залежності від конкретного пристрою. Найбільш відомі приклади блокових пристроїв – магнітні диски і магнітні стрічки. Для диска поняття блоку звичайно збігається з поняттям сектора. Зокрема, для IBM-сумісних ПК сектор (блок) диска дорівнює 512 байт.

Блокова архітектура обумовлена особливостями використовуваного середовища і, крім того, блокове введення/висновок більш ефективний для високошвидкісних пристроїв, оскільки при цьому зменшується відносна частка часу, що витрачається на підготовчі і заключні операції при кожнім звертанні до пристрою.

47* ***Фізичні***, ***логічні*** і ***віртуальні*** пристрої. Під фізичним пристроєм звичайно розуміється деякий реально існуючий прилад, «залозка». Насправді, з погляду програмної архітектури для наявності фізичного пристрою досить знати набір адрес, команд, переривань і інших сигналів, що дозволяють виконувати операції з даними. Куди йдуть або відкіля приходять ці сигнали – це питання, що не стосується програміста.

Логічний пристрій – це поняття, що характеризує спеціальне призначення пристрою в даної ОС. Наприклад, «завантажувальний диск» (тобто той, з якого була виконана завантаження ОС). Найбільш важливими логічними пристроями в багатьох ОС є ***пристрій стандартного введення*** і ***пристрій стандартного висновку***. Їх можна спрощено визначити як пристрою, використовувані для введення і, відповідно, висновку «за замовчуванням», тобто коли в програмі явно не зазначені інший пристрій або файл для введення/висновку. Як правило, для сучасних комп'ютерів пристроєві стандартного уведення відповідає фізичний пристрій – клавіатура, а пристроєві стандартного висновку – монітор. Важливо, однак, розуміти, що ця відповідність може бути змінено: стандартний висновок може бути перепризначений, наприклад, на принтер або у файл, стандартне введення – на вилучений термінал, на файл і т.п.

Поняття «віртуальний» у програмуванні, узагалі говорячи, означає приблизно наступне: «щось, що насправді не існує, але ведуче себе так, ніби воно існувало». З цього погляду, віртуальний пристрій – це програмно реалізований об'єкт, що поводиться подібно деякому фізичному пристроєві, хоча насправді використовує ресурси зовсім інших пристроїв (або навіть ніяких пристроїв). Приклади віртуальних пристроїв досить різноманітні:

48* віртуальні диски, розташовані насправді в оперативній пам'яті (такі пристрої були популярні наприкінці 80-х років);

49* віртуальна пам'ять, розташована насправді на диску;

50* віртуальні CD і DVD – програми, що імітують поводження відповідних пристроїв;

51* віртуальний екран, наданий DOS-програмі, що працює в режимі вікна Windows (програма працює так, ніби їй був наданий весь екран, але насправді система направляє висновок програми у відведене їй вікно);

52* самий забавний (але дуже корисний) приклад – порожній пристрій, якому не відповідає ніяка апаратура. Чому його взагалі можна назвати пристроєм? Тому що відповідна системна програма (драйвер порожнього пристрою) коректно виконує всі дії, що зобов'язаний виконувати драйвер пристрою. Такий пристрій безвідмовно приймає вихідний потік символів (і відразу викидає прийняті дані), а також може використовуватися для введення, але при цьому відразу повідомляє – мов, досягнуть кінець файлу. Порожній пристрій корисно в тих випадках, коли деяка програма вимагає неодмінно указати файл або пристрій для висновку об'ємних і не дуже потрібних даних. Крім того, копіювання файлу на порожній пристрій – це простий спосіб переконатися, що

файл читається без помилок.

2.3. Переривання

Переривання (апаратні) – це сигнали, при надходженні яких нормальна послідовність виконання програми може бути перервана, при цьому система запам'ятовує інформацію, необхідну для поновлення роботи перерваної програми, і передає керування ***підпрограмі обробки переривання*** (ISR, Interrupt Service Routine). По завершенню обробки, як правило, керування повертається перерваній програмі.

Усі переривання можна розділити на три основних типи:

53* апаратні переривання від периферійних пристроїв;

54* внутрішні апаратні переривання (називані також ***виключеннями***, exceptions);

55* програмні переривання.

У переважній більшості ОС обробок усіх переривань бере на себе сама система, оскільки це занадто «інтимна» частина роботи, здатна вплинути на функціонування всіх системних і прикладних програм.

Оскільки типи і різновиди переривань досить різноманітні і кожний з них вимагає особливої обробки, більшість процесорів підтримує ***векторні переривання***. Це означає, що кожен різновид переривання має свій номер, і цей номер використовується як індекс у масиві, що зберігає адреси ISR для всіх переривань. При виникненні переривання апаратура комп'ютера по номері переривання визначає адресу підпрограми обробки і викликає неї.

Для того щоб деякі найбільш відповідальні ділянки системних програм виконувалися без переривань, система має можливість тимчасово заборонити прийом більшості переривань. Така заборона повинна встановлюватися лише на короткі інтервали часу, не більш декількох миллисекунд.

Програмні переривання викликаються виконанням спеціальної команди, але обробляються точно так само, як інші типи переривань. По суті, команда програмного переривання являє собою особливий випадок виклику підпрограми, але при цьому замість адреси підпрограми вказується номер переривання, оброблювач якого повинний бути викликаний. У більшості сучасних ОС програмних переривань використовуються для переходу з режиму користувача в режим ядра при виклику системних функцій із прикладної програми.

Одним з найважливіших джерел переривань є периферійні пристрої. Як правило, пристрій генерує сигнал переривання в одному з двох випадків:

56* при переході в стан готовності;

57* при виникненні помилки виконання операції.

Стан готовності – це такий стан пристрою, у якому воно готово прийняти і виконати команди від процесора. Для пристрою введення готовність означає наявність у пристрої даних, що можуть бути передані в процесор (наприклад, клавіатура переходить у стан «Готовий» при натисканні клавіші і повертається в стан «Не готове», коли код натиснутої клавіші лічений у процесор). Для пристрою висновку готовність – це можливість прийняти від процесора дані, які варто вивести. Наприклад, матричний принтер приймає символи, які потрібно надрукувати, у свій внутрішній буфер. Якщо буфер повний, принтер переходить у стан «Не готове» доти, поки частина символів буде надрукована й у буфері звільниться місце. Дисковий нагромаджувач при початку виконання нової операції читання або записи на диск переходить у стан «Не готове», а після завершення операції повертається в стан «Готове». У кожнім з цих випадків перехід у стан «Готовий» – це привід для пристрою нагадати про себе процесорові: звернете на мене увага, я до

ваших послуг! Для цього і служить сигнал переривання.

Помилка операції також вимагає втручання системи або користувача. Наприклад, при помилці відсутності папера в лотку принтера система повинна сповістити про це користувача; при помилці читання з диска або система, або користувач повинний вирішити, що робити: повторити операцію, завершити програму або продовжити виконання.

Не кожен пристрій генерує переривання. Наприклад, монітор ПК не видає переривань: він «завжди готовий», тобто завжди може прийняти дані для відображення, і він «ніколи не помиляється», точніше сказати, його несправність виявляється «на око».

2.4. Архітектура підсистеми введення/висновку

З програмної точки зору, пристрій (або його контролер) звичайно представлений одним або декількома ***регістрами***. Регістр пристрою – це адресуемое машинне слово, використовуване для обміну даними або сигналами між пристроєм і процесором. Можна виділити два основних типи регістрів.

58* ***Регістр даних*** служить для обміну даними. Запис даних у такий регістр (якщо вона можлива) означає висновок даних на пристрій, читання даних з регістра – уведення з пристрою.

59* ***Регістр керування і стану*** містить два типи двоичных розрядів (битов). Біти стану служать для передачі процесорові інформації про поточний стан пристрою (наприклад, прапорів готовності і помилки, сигналів переривання). Біти керування служать для передачі на пристрій команд, що дозволяють задати виконувану операцію, запустити виконання операції, установити режими роботи пристрою і т.п.

У різних комп'ютерах використовується один із двох способів адресації регістрів пристроїв.

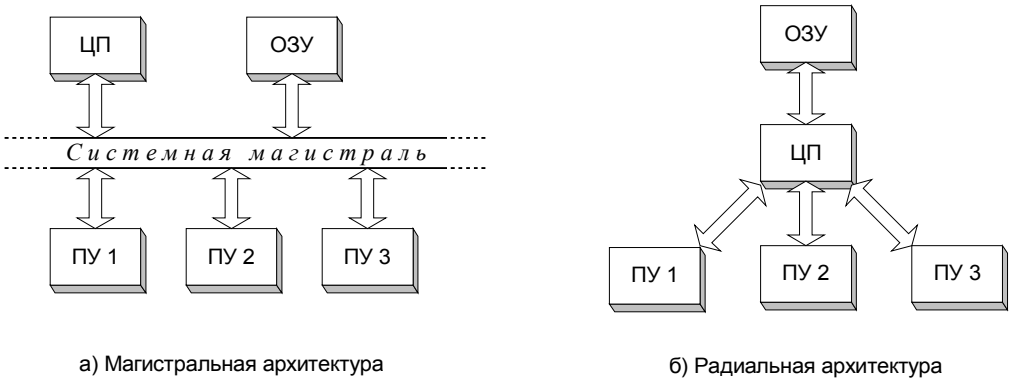
60* Відображення регістрів пристроїв на пам'ять. При цьому способі для пристроїв приділяється визначена частина адресного простору пам'яті, а для роботи з пристроями можна використовувати ті ж команди, що і для роботи з основною пам'яттю (наприклад, команду **`MOV`**).

61* Адресація регістрів через порти введення/висновку. Для портів приділяється окремий адресний простір, і для роботи з ними маються спеціальні команди (наприклад, **`IN`** і **`OUT`**).

Перший спосіб зручніше для програмування, оскільки дозволяє використовувати більш широкий набір команд. Однак цей спосіб сутужніше реалізувати на апаратному рівні, оскільки апаратура повинна визначати, чи відноситься конкретна адреса до пам'яті або до пристрою, і по-різному обробляти ці два випадки.

Серед різних можливих конфігурацій однопроцесорної обчислювальної системи прийнято виділяти два основних типи: системи з магістральної і з радіальною архітектурою (мал. 2- 1).

Магистральная и радиальная архитектура



а) Магистральная архитектура

б) Радиальная архитектура

Рис. 2- 1

62* ***Магістральна*** архітектура заснована на підключенні всіх наявних пристроїв, включаючи процесор і пам'ять, до єдиної системної магістралі (шині), що поєднує в собі лінії передачі даних, адрес і керуючих сигналів. Спільне використання магістралі різними пристроями підкоряється спеціальним правилам (протоколові), що забезпечує коректність роботи магістралі.

63* ***Радіальна*** архітектура припускає, що кожне з пристроїв, включаючи пам'ять, підключається до процесора окремо, незалежно від інших пристроїв, і взаємодіє з процесором за власними правилами.

Для програміста поняття магістральної і радіальної архітектури мають трохи інший зміст, чим для інженер-системотехніка. З погляду програмної архітектури, неважливо, чи приєднаний пристрій до процесора прямо або за посередництвом системної магістралі. Важливо те, які сигнали повинний посилати і приймати програма, що працює з пристроєм, і які команди можуть для цього використовуватися.

Основна особливість магістральної архітектури – однаковий спосіб підключення всіх пристроїв. Структура регістрів пристрою стандартизується, при цьому визначається, якими сигналами будь-який пристрій може обмінюватися з процесором і якими розрядами регістра повинні відповідати ці сигнали. Звичайно, не всякий пристрій має потребу у використанні всього набору стандартних сигналів. Деякі типи пристроїв можуть, наприклад, не генерувати переривань, не повідомляти про помилки. Але ті сигнали, що пристрій використовує, повинні відповідати стандартові даної магістралі.

Перевагою магістральної архітектури є простота підключення нових типів пристроїв, тому така архітектура особливо зручна для відкритих обчислювальних систем, тобто таких, котрі розраховані на розширюваний набір периферійних пристроїв.

Навпроти, для радіальної архітектури характерний індивідуальний вибір способу підключення, найбільш зручного для кожного типу пристроїв. При цьому в принципі можна досягти економії апаратних ресурсів і більш високої ефективності. Трапляється навіть, що в одному порту поєднуються керуючі сигнали від декількох різних пристроїв. Очевидно, подібна архітектура зручна тільки в тому випадку, коли вона розрахована на постійний набір пристроїв. Розширення радіальної системи завжди викликає утруднення.

Виходячи них цих визначень, не так вуж легко точно охарактеризувати сучасні IBM-сумісні ПК. Вихідна модель IBM PC мала досить чітко виражену радіальну архітектуру і невеликий набір стандартних пристроїв. У наступних моделях були зроблені значні кроки по стандартизації підключення нових пристроїв. Однак і сьогодні ці комп'ютери не тягнуть на магістральну архітектуру в повному розумінні слова: у них для цього занадто багато різних шин.

Важливою деталлю архітектури сучасних комп'ютерів є такий пристрій, як контролер ***прямого доступу до пам'яті*** (ПДП, англ. DMA – Direct Memory Access). Якщо звичайно весь обмін даними йде через регістри процесора, то ПДП має на увазі пряме перенесення даних із пристрою в пам'ять або назад. Роль процесора в даному випадку тільки в тім, щоб ініціювати операцію введення/висновку блоку даних, пославши відповідні команди контролерові ПДП. Далі процесор не бере участь у виконанні обміну даними. Завершивши операцію, контролер ПДП посилає сигнал переривання, сповіщаючи про це процесор. Це дозволяє підвищити продуктивність системи за рахунок часткового розвантаження процесора і магістралі.

2.5. Способи організації введення/висновку

2.5.1. Уведення/висновок по опитуванню і по перериваннях

Розглянемо більш докладно роботу програми, що безпосередньо виконує введення або висновок даних на конкретний пристрій. (Насправді, цією роботою звичайно займається драйвер пристрою, так що ми фактично розглядаємо логіку роботи драйвера.)

Для визначеності покладемо, що програма повинна видати **N** байт даних з масиву **A** на символьний пристрій **X**. Для операції введення можуть використовуватися ті ж підходи, що будуть розглянуті тут для операції висновку.

Нехай архітектура пристрою представлена регістром даних **X.DATA** і прапором готовності **X.READY**. Коли **X.READY** = **TRUE**, у регістр **X.DATA** можна видавати черговий байт даних. Запишемо на псевдокоді, близькому до мови Паскаль, варіанти організації відповідної програми.

а) Уведення/висновок без перевірки готовності

```
i := 1;
while i <= N do begin
  X.DATA := A[i];
  i := i + 1;
end;
```

Цей «нахабний» спосіб висновку цілком працездатний, якщо використовується «завжди готове» пристрій (наприклад, монітор), тобто прапор **X.READY** завжди щирий і тому взагалі не потрібний. При спробі використовувати той же підхід для висновку на принтер ми переконалися б, що надруковано будуть лише деякі символи, яким пощастило бути виданими в рідкі моменти готовності принтера.

б) Уведення/висновок по опитуванню готовності

```
i := 1;
while i <= N do begin
  while not X.READY do
    ;
  X.DATA := A[i];
  i := i + 1;
end;
```

Тут доданий цикл чекання, у якому не робиться нічого, крім постійної циклічної перевірки готовності пристрою. Передача даних відбувається тільки тоді, коли пристрій готовий. Оскільки після видачі одного байта пристрій цілком може знову перейти в стан неготовності, варто знову виконувати цикл чекання, поки виданий символ не буде оброблений пристроєм.

Така організація введення/висновку дозволяє коректно працювати з будь-якими пристроями. Цей спосіб дійсно застосовується в деяких однозадачных системах. Недоліком даного способу є непродуктивна витрата часу на постійне «довбання» прапора готовності. При

сучасному співвідношенні швидкостей роботи процесора і периферії, цикл чекання може повторюватися мільйони разів перед видачею кожного байта. Більш того, якщо з якихось причин пристрій узагалі не перейде в стан готовності, то робота всієї системи може бути паралізована нескінченним циклом чекання.

в) Уведення/висновок по перериваннях

```
     i := 1;
     while i <= N do begin
X_INT: if not X.READY
         return;
       X.DATA := A[i];
       i := i + 1;
     end;
```

Тут зник цикл чекання, замість нього – однократна перевірка готовності й оператор повернення, якщо не готово.

Куди, власне, відбувається повернення? Щоб це зрозуміти, треба згадати, що даний фрагмент – явно не єдина програма, що працює в даний момент на ЕОМ. Очевидно, операція висновку була почата операційною системою по запиті якоїсь програми. Даний фрагмент був викликаний як підпрограма ОС, і повернення означає передачу керування ОС. Як система розпорядиться отриманим часом? Це вже зовсім інше питання, не зв'язаний із уведенням/висновком. Наприклад, ОС може переключитися на інший процес. Або, від чогось робити, запустити екранну заставку або програму самотестування.

Але як же бути з кинутої на полпути операцією висновку? Для її поновлення буде використане апаратне переривання, що повинне видати пристрій **X** при переході в стан готовності. Системний оброблювач переривання повинний буде передати керування за адресою, позначеному міткою **X_INT**. Після незайвої додаткової перевірки готовності програма висновку передасть черговий байт на пристрій, потім знову перевірить готовність і, можливо, знову поверне керування системі. Таким чином, виконання введення/висновку розбивається на окремі інтервали роботи при готовності пристрою, що перемежовуються роботою системи, поки пристрій не готовий.

Для пристроїв, що використовують контролер ПДП, можливі варіанти організації роботи залишаються, по суті, тими ж, але тільки використовуються набагато більш великі операції: замість введення або висновку одного елемента даних виконується введення/висновок цілого блоку даних, і тільки після цього контролер переходить у стан готовності і генерує переривання.

2.5.2. Активне і пасивне чекання

Поговоримо докладніше про одне важливе розходження між способами введення/висновку по опитуванню готовності і по перериваннях.

Основною особливістю введення/висновку по опитуванню готовності є цикл чекання. Якщо виконується введення або висновок на повільний пристрій (наприклад, матричний принтер), то цей скромно виглядає цикл буде займати, м'яко говорячи, 99% усього часу роботи процесора. Якщо відбувається чекання введення з клавіатури, то процесор узагалі не буде робити нічого корисного, поки користувач не удосужится натиснути клавішу.

Таке абсурдне використання процесора може бути виправдано хіба що в тому випадку, якщо для нього немає ніякої більш корисної роботи. Це можливо у випадку однозадачной ОС, коли працююча прикладна програма не може просуватися далі, поки не довершена операція введення/висновку. У цьому випадку введення/висновок по опитуванню не позбавлений визначених достоїнств: він не зв'язаний з обробкою переривань, що вимагає деякого часу, а тому сповільнює реакцію на перехід пристрою в стан готовності.

Спосіб чекання програмою деякої події, заснований на постійній циклічній перевірці очікуваної умови, називається ***активним чеканням*** (busy waiting). Це поняття застосовується не тільки стосовно введення/висновкові, але й у багатьох інших ситуаціях, що виникають при роботі системних і прикладних програм.

Якщо розглядається многозадачная ОС, у якій може бути кілька активних задач одночасно, то активне чекання стає зовсім неприйнятним. У цьому випадку витрата процесорного часу на виконання циклічного опитування завдає прямої шкоди іншим програмам, що могли б використовувати цей час більш осмислено. Тому при розробці многозадачных систем, як при введенні/висновку, так і в деяких інших ситуаціях, обов'язково реалізується ***пасивне чекання***, тобто така реалізація чекання, при якій програма, що очікує, не затрачає процесорного часу. Для реалізації пасивного чекання завжди в тій або іншій формі використовуються апаратні переривання. Приватним прикладом пасивного чекання є розглянутий вище введення/висновок по перериваннях.

2.5.3. Синхронне й асинхронне введення/висновок

Програміст, що розробляє прикладні програми, не повинний думати про такі речі, як спосіб роботи системних програм з регістрами пристроїв. Система ховає від додатків деталі низкоуровневой роботи з пристроями. Однак розходження між організацією введення/висновку по опитуванню і по перериваннях знаходить визначене відображення і на рівні системних функцій, у виді функцій для синхронного й асинхронного введення/висновку.

Виконання функції ***синхронного введення/висновку*** містить у собі запуск операції введення/висновку і чекання завершення цієї операції. Тільки після завершення введення/висновку функція повертає керування програмі, що викликав.

Синхронне введення/висновок – це найбільш звичний для програмістів спосіб роботи з пристроями. Стандартні процедури введення/висновку мов програмування працюють саме таким способом.

Виклик функції ***асинхронного введення/висновку*** означає тільки запуск відповідної операції. Після цього функція відразу повертає керування програмі, що викликав, не чекаючи завершення операції.

Розглянемо, наприклад, асинхронне введення даних. Зрозуміло, що програма не може звертатися до даних, поки немає впевненості, що їхнє введення довершене. Але цілком можливо, що програма може поки що зайнятися іншою роботою, а не простоювати в чеканні.

Рано або пізно програма все-таки повинна приступити до роботи з уведеними даними, але попередньо переконатися, що асинхронна операція вже завершилася. Для цього різні ОС надають кошти, які можна розбити на три групи.

64* Чекання завершення операції. Це як би «друга половина синхронної операції». Програма спочатку запустила операцію, потім виконала якісь сторонні дії, а тепер чекає закінчення операції, як при синхронному введенні/висновку.

65* Перевірка завершення операції. При цьому програма не очікує, а тільки перевіряє стан асинхронної операції. Якщо введення/висновок ще не довершений, то програма має можливість ще якийсь час погуляти.

66* Призначення процедури завершення. У цьому випадку, запускаючи асинхронну операцію, програма користувача вказує системі адреса користувальницької процедури або функції, що повинна бути викликана системою після завершення операції. Сама програма може більше не цікавитися ходом уведення/висновку, система нагадає їй про це в потрібний момент, викликавши зазначену функцію. Цей спосіб найбільш гнучкий, оскільки в процедурі завершення користувач може передбачити будь-які дії.

У Windows прикладній програмі доступні всі три способи завершення асинхронних операцій. У UNIX асинхронних функцій уведення/висновку ні, однак той же ефект асинхронности може бути досягнуть інакше, шляхом запуску додаткового процесу.

Асинхронне виконання введення/висновку дозволяє в деяких випадках підвищити продуктивність роботи і забезпечити додаткові функціональні можливості. Без такої найпростішої форми асинхронного введення, як «уведення з клавіатури без чекання», були б неможливі численні комп'ютерні ігри і тренажери. У той же час логіка програми, що використовує асинхронні операції, складніше, ніж при синхронних операціях.

А в чому полягає згадана вище зв'язок між синхронними/асинхронними операціями і способами організації введення/висновку, розглянутими в попередньому пункті? Відповісти самі на це питання.

2.6. Буферизация і кэширование

2.6.1. Поняття буферизации

Буферизацию в самому широкому змісті можна визначити як таку організацію введення/висновку, при якій дані не передаються безпосередньо з пристрою в задану область пам'яті (або з області пам'яті на пристрій), а попередньо направляються в допоміжну область пам'яті, називану ***буфером***. Як правило, организуемые системою буфери невидимі для прикладного програміста, він одержує дані як готовий результат. Нерідко дані «по дорозі» проходять через кілька буферів різного призначення.

Існує кілька причин для використання буферизации, найважливіші з яких розглянуті нижче.

2.6.2. Згладжування нерівномірності швидкостей процесів

Досить часто в роботі ОС зустрічається ситуація, коли один процес породжує дані, що повинні оперативно оброблятися іншим процесом. Як приклад можна привести прийом по мережі даних, що повинні оброблятися браузером або іншою прикладною програмою.

Швидкість прийому даних дуже нерівномірнаі: інтервали часу інтенсивного надходження даних перемежовуються з інтервалами простою. Обробка даних прикладною програмою теж не обов'язково йде з постійною швидкістю. У результаті, хоча середня швидкість обробки може бути цілком достатньої, не виключено, що в деякі моменти обробна програма буде «захлинатися» даними. Це може привести до втрати частини даних, що не встигли пройти обробку.

Стандартним рішенням у цій ситуації є використання буфера, розмір якого досить великий, щоб умістити всі дані, що очікують обробки. Чим більше буфер, тим менше імовірність утрати даних через його переповнення.

2.6.3. Распараллеливание введення й обробки

У багатьох обчислювальних системах маються апаратні можливості сполучити в часі виконання операцій уведення/висновку й обробку даних процесором. Щоб використовувати ці можливості, дані при введенні направляються в буфер. Після заповнення буфера його дані пересилаються в обробну програму, а їхня обробка виконується паралельно з нагромадженням наступної порції даних у буфері.

Ще більш ефективна схема роботи з двома буферами, що переключаються. Поки в першому буфері накопичуються дані, що вводяться, попередня порція даних обробляється в другому буфері, без утрати часу на пересилання. Потім буфери міняються ролями: у першому буфері обробляється наступна введена порція даних, а другий буфер використовується для введення, і т.д.

Аналогічним образом буферизация може використовуватися і при висновку даних.

У деяких випадках виявляється вигідно виконувати введення «з випередженням», тобто вводити ті дані, що поки не запитані обробним процесом, але по всій імовірності незабаром знадобляться.

2.6.4. Узгодження розмірів логічного і фізичного запису

Логічним записом називають порцію даних, зазначений в операторі введення/висновку. Розмір логічного запису визначається логікою роботи програми або, наприклад, логічною структурою бази даних.

При фактичному виконанні читання або запису на блоковий пристрій обробляється порція даних, називаний фізичним записом або блоком. Розмір фізичного запису визначається особливостями пристрою (для диска це один сектор) і ніяк не зв'язаний з логікою програми.

На мал. 2- 2 показана ситуація, коли логічний запис містить 100 байт, а фізична – 512 байт.

Логические и физические записи

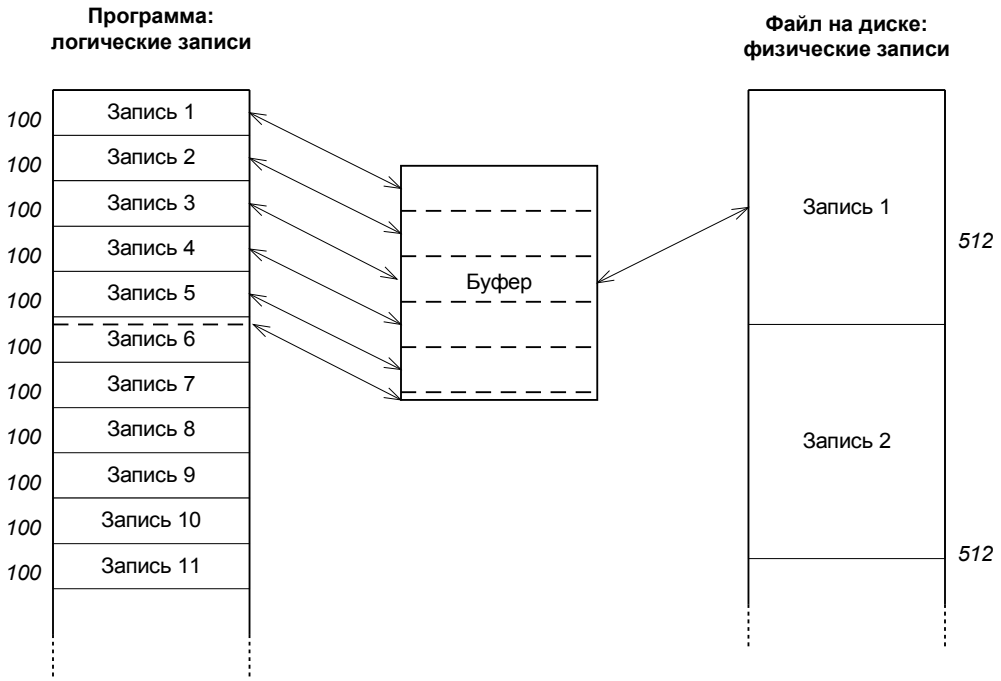


Рис. 2- 2

Припустимо, що логічні записи в послідовному порядку записуються у файл на диску. Якщо кожен оператор висновку логічного запису буде викликати негайний запис на диск, то видача перших п'яти логічних записів зажадає п'ять разів виконати послідовність операцій: «читання фізичного запису з диска – зміна 100 байт – запис зміненого запису на диск». На шостий раз вийде ще гірше, оскільки прийдеться читати – змінювати – записувати не одну, а два фізичних записи.

Використання буфера для нагромадження даних до розміру фізичного запису дозволяє різко скоротити кількість операцій запису на диск і майже цілком виключити читання з диска.

2.6.5. Редагування при інтерактивному введенні

Трохи особняком від інших форм буферизации коштує використання буфера рядка при

введенні з клавіатури.

Для користувача звично, що в процесі введення числових або строкових значень він може легко відкоригувати помилки введення: «забити» невірний символ, повернутися в будь-яке місце рядка, що вводиться, і внести там зміни і т.п. При цьому прикладна програма «не бачить» процесу редагування рядка, вона одержує весь рядок цілком після натискання, наприклад, клавіші **`Enter`**. Щоб забезпечити можливість редагування рядка, що вводиться, використовується буфер рядка, виділюваний або ОС, або бібліотекою часу виконання конкретної системи програмування. Усе редагування виконується над символами, що містяться в цей буфер підпрограмами введення з клавіатури. Після натискання **`Enter`** відбувається або копіювання символів з буфера в масив, виділений прикладною програмою, або передача цій програмі покажчика на буфер.

2.6.6. Кэширование дисків

Дуже важливою, специфічною формою буферизации є ***кэширование*** . Цей термін означає використання порівняно невеликої по обсязі, але швидкодіючої пам'яті для того, щоб зменшити кількість звертань до більш повільної пам'яті великого обсягу.

Ідея кэширования ґрунтується на так називаній ***гіпотезі про локальність посилань***. Ця гіпотеза полягає в наступному. Якщо в якийсь момент часу відбулося звертання до визначеної ділянки даних, то найближчим часом можна з високою імовірністю очікувати повторення звертань до тих же самим даним або ж до сусідніх ділянок даних. Звичайно, локальність посилань не можна вважати законом, однак практика показує, що ця гіпотеза виправдується для гнітючої більшості програм.

У сучасних обчислювальних системах може використовуватися кілька рівнів кэширования. У даному курсі не розглядається апаратний кэш процесора, що дозволяє скоротити число звертань до основної пам'яті за рахунок використання швидкодіючих регістрів. До роботи ОС більш пряме відношення має програмне кэширование пристроїв довільного доступу (дискових нагромаджувачів). У цьому випадку гіпотезу про локальність посилань можна переформулювати більш конкретно: якщо програма виконала читання або запис даних з деякого блоку диска, те досить імовірно, що у швидкому майбутньому підуть ще операції читання або запису даних з того ж блоку.

У ролі швидкодіючої пам'яті (кэша) тут виступає масив буферів, розміщений у системній пам'яті. Кожен буфер складається з заголовка і блоку даних, що відповідає по розмірі блокові (секторові) диска. Заголовок буфера містить адреса блоку диска, копія якого в даний момент утримується в буфері, і кілька прапорів, що характеризують стан буфера.

Коли система одержує запит на читання або запис визначеного блоку даних диска, він насамперед перевіряє, чи не утримується в даний момент копія цього блоку в одному з буферів кэша. Для цього потрібно виконати пошук по заголовках буферів. Якщо блок знайдений у кэше, то звертання до диска виконуватися не буде. Замість цього дані читаються з буфера або, відповідно, записуються в буфер. У випадку запису даних випливає також у заголовку буфера відзначити за допомогою спеціального прапора, що буфер став «***брудним***», тобто його вміст не відповідає даним на диску.

Якщо необхідний блок диска не знайдений у кэше, то для нього повинний бути виділений буфер. Проблема в тім, що загальна кількість буферів кэша обмежено. Щоб віддати один з них під необхідний блок, треба «витиснути» з кэша один із блоків, що там зберігалися. При цьому, якщо блок, що витісняється, «брудний», те він повинний бути «очищений», тобто записаний на диск. При витисненні «чистого» блоку ніяких операцій з диском виконувати не треба.

Який із блоків, що зберігаються в кэше, варто вибрати для витиснення, щоб скоротити

загальна кількість звертань до диска? Це украй важливе питання, і якщо він буде вирішуватися неправильно, те вся робота системи може загальмуватися через постійні звертання до диска.

Мається теоретично оптимальне рішення даної задачі, що полягає в наступному. Число звертань до диска буде мінімально, якщо щораз вибирати для витиснення той блок даних, до якого в майбутньому довше всього не буде звертань. На жаль, скористатися цим правилом на практиці неможливо, тому що послідовність звертань до блоків диска непередбачена. Даний теоретичний результат корисний тільки як недосяжний ідеал, з яким можна порівнювати результати застосування більш реалістичних алгоритмів вибору.

Серед алгоритмів, використовуваних на практиці, кращим вважається алгоритм ***LRU*** (Least Recently Used, у вільному перекладі «давно що не використовувався»). Він полягає в наступному: вибирати для витиснення випливає той блок, до якого довше всього не було звертань. Тут саме використовується принцип локальності посилань: раз звертань давно не було, те, імовірно, їх і не буде найближчим часом.

Як на практиці реалізується вибір блоку за правилом LRU? Очевидне рішення – при кожнім звертанні до буфера записувати в його заголовку поточний час, а при виборі для витиснення шукати самий ранній запис – занадто громіздко і повільно. Є набагато краща можливість.

Усі буферы кэша зв'язуються в лінійний список. У заголовку кожного буфера зберігається посилання на наступний один по одному списку буфер (фактично зберігається індекс цього буфера в масиві буферів). При кожнім звертанні до блоку даних для читання або запису виконується також переміщення відповідного буфера в кінець списку. Це не означає переміщення даних, що зберігаються в буфері, змінюються тільки кілька посилань у заголовках.

У результаті постійного переміщення використаних блоків у кінець списку буферів цей список виявляється відсортованим по зростанню часу останнього звертання. На початку списку виявляється той буфер, до даних якого довше всього не було звертань. Він^-те нам і потрібний як кандидат на витиснення.

На мал. 2- 3 показаний масив буферів, зв'язаний у список.

LRU-список буферов

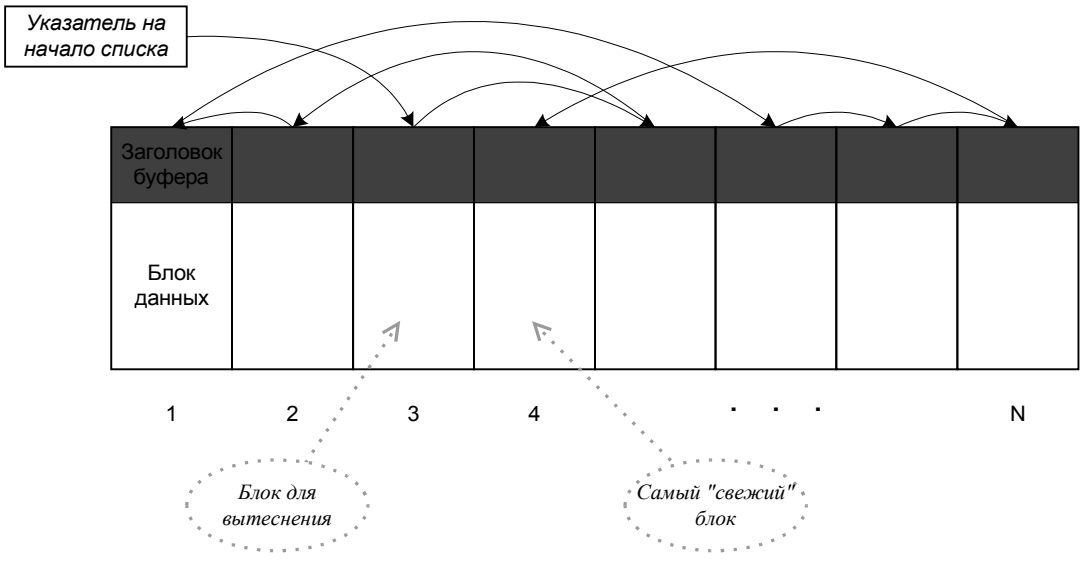


Рис. 2- 3

Тепер про «брудні» буфери. У яких випадках повинна виконуватися їх «очищення», тобто запис блоку даних з кэш-буфера на диск? Можна назвати три таких випадки.

67* Вибір блоку для витиснення з кэша.

68* Закриття файлу, до якого відносяться «брудні» блоки. Загальноприйнято, що при закритті файлу повинне виконуватися його збереження на диску.

69* Операція примусового очищення всіх буферів або тільки буферів, що відносяться до визначеного файлу. Подібна операція може виконуватися для підвищення надійності збереження даних, як страховка від можливих збоїв. В ОС UNIX, наприклад, очищення всіх буферів традиційно виконується кожні 30 с.

Варто визнати, що кэширование операцій запису на диск, на відміну від кэширования читання, завжди створює визначену небезпеку втрати даних. У випадку випадкового збою системи, відключення харчування і т.п. може виявитися, що важлива інформація, которую випливало записати на диск, застрягла в брудних буферах кэша і була тому загублено. Це неминуча плата за значне підвищення продуктивності системи. Програми, що вимагають високої надійності роботи з даними (наприклад, банківські програми), звичайно записують дані прямо на диск. При цьому кэш або не використовується взагалі, або в кэш-буфер заноситься копія даних, що може придатися при наступних операціях читання.

«Вузьким місцем» кэширования дисків є пошук необхідного блоку даних у кэше. Як було описано вище, для цього система переглядає заголовки буферів. Якщо кэш складається з декількох сотень буферів, час пошуку буде відчутно. Один з можливих прийомів прискорення пошуку, використовуваний у UNIX, показаний на мал. 2- 4.

Структура дискового кэша UNIX

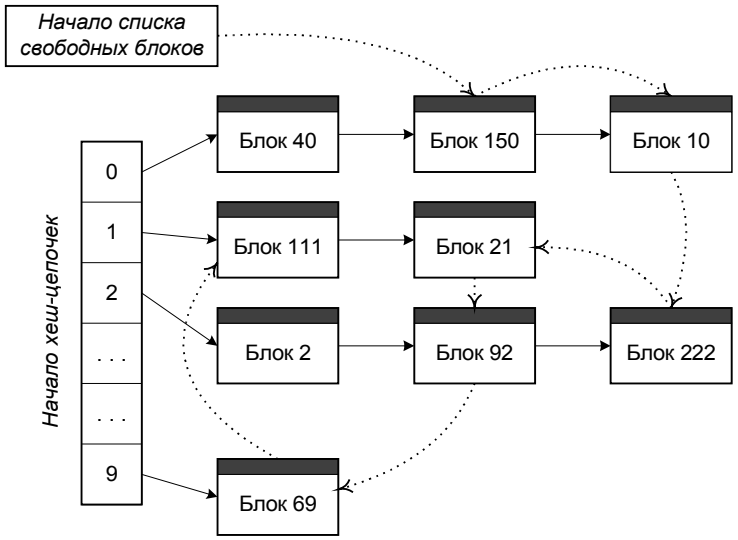


Рис. 2- 4

У UNIX кожен кэш-буфер може входити одночасно в два лінійних списки. Один з них, називаний «списком вільних блоків», це знайомий нам LRU-список, використовуваний для визначення блоку, що підлягає витисненню. Слово «вільний» не значить «порожній»; у даному випадку це слово означає блок, не зайнятий у сучасний момент в операції читання/запису, виконуваної яким-небудь процесом. Інший список називається «хеш-цепочкой» і використовується для прискорення пошуку потрібного блоку.

При записі в буфер даних, що відповідають деякому блокові диска, номер хеш-цепочки, у

которую буде поміщений цей буфер, визначається як залишок від розподілу номера блоку на N – кількість хеш-цепочек. Для наочності на малюнку прийняте значення N = 10. Таким чином, блоки з номерами 120, 40, 90 попадають у ланцюжок 0, блоки 91, 1, 71 – у ланцюжок 1 і т.д. Коли система шукає в кэше блок з визначеним номером, вона насамперед по номері блоку визначає, у який з хеш-цепочек цей блок повинний знаходитися. Якщо блоку немає в цьому ланцюжку, то його взагалі немає в кэше. Таким способом удається скоротити пошук у кращому випадку в N раз (це якщо всі ланцюжки виявляться однакової довжини).

Переміщення буфера з однієї хеш-цепочки в іншу, як і його переміщення в кінець списку вільних блоків, не вимагає перезапису всього блоку даних у пам'яті і виконується шляхом зміни посилань у заголовках блоків.

Ще одна особливість кэшированія дисків у UNIX полягає в тому, що при виявленні на початку списку вільних блоків «брудних» буферів система запускає процеси їхнього очищення, але не чекає завершення цих процесів, а вибирає для витиснення перший за списком чистий блок. Після завершення очищення блоки повертаються в початок списку вільних блоків, залишаючись першими кандидатами на витиснення.

2.6.7. Випереджальне читання.

У тому випадку, якщо обробка даних ведеться послідовним образом (від початку файлу до кінця), кэширование не дає значного ефекту. Після того, як оброблені дані з одного блоку, подальше перебування цього блоку в кэш-буфере даремно. Значно більш корисної в цьому випадку може виявитися інша спеціальна форма буферизации, відома як ***випереджальне читання***. Вона полягає в тім, що при звертанні до деякого блоку диска система, виконавши читання необхідного блоку, зчитує потім ще трохи наступних за ним блоків. Якщо апаратура дозволяє виконувати операцію читання одночасно з обробкою раніше прочитаних даних, то велико імовірність, що до моменту, коли наступний блок даних буде запитаний для обробки, цей блок уже виявиться прочитаним.

Як правило, системі невідомо, чи буде обробка файлу вестися в режимі послідовного або довільного доступу, тому часто використовується та або інша комбінація кэшированія з випереджальним читанням. У Windows програма, що відкриває файл, може вказати системі, для якого способу доступу бажано оптимизировать механізм буферизации.

Ідея випереджального читання одержала цікавий розвиток у Windows XP. У цій системі введений механізм випереджального завантаження даних (prefetch), що заснований на автоматичному зборі і збереженні статистики про те, які файли і каталоги використовуються в ході завантаження ОС і при запуску конкретних додатків, а також які дані читаються з цих файлів у перші хвилини роботи. При наступних завантаженнях ОС і запусках додатків система виконує очікувані операції читання ще до того, як вони будуть у дійсності запитані компонентами, що завантажуються, ОС або додатком. При цьому система планує порядок операцій таким чином, щоб скоротити переміщення читаючих голівок і тим самим прискорити завантаження даних.

2.7. Драйвери пристроїв

Драйвер пристрою – це системна програма, що під керуванням ОС виконує всі операції з конкретним периферійним пристроєм. Драйвер є як би посередником між ОС і пристроєм. Перед драйверами коштують дві однаково важливі, але важко сумісні задачі:

70* забезпечити можливість стандартного звертання до будь-якого пристрою, ховаючи від інших частин ОС специфічні особливості окремих пристроїв;

71* домогтися максимально ефективного використання усіх функціональних можливостей і

особливостей конкретних пристроїв.

Можливість стандартними засобами працювати з різними пристроями дуже бажана з погляду архітектури ОС і зручності програмування. Було б вкрай огидно, якби при написанні прикладної програми потрібно було заздалегідь враховувати, яка модель принтера буде використовуватися для видачі результатів. Навпаки, у більшості випадків прикладний програміст навіть не повинний знати, чи буде це принтер або плоттер-графопостроитель, або ж результати будуть відображатися на екрані. Великі проблеми могли б виникнути і при заміні однієї моделі принтера, диска, монітора на іншу, якби така заміна зажадала переписувати заново всі програми, що працюють з цим пристроєм. Інша справа, якщо всі особливості пристрою враховуються в одним-єдиному місці, а саме – у драйвері цього пристрою.

Зрозуміло, цілком сховати всі розходження між пристроями неможливо. Ніяким образом не можна дорівняти, скажемо, диск до клавіатури, і навіть різні типи дисків схожі, але не зовсім. Наприклад, для дискет можна виконати таку операцію, як перевірка зміни носія (фактично при цьому перевіряється, чи відкривалася кишеня дисководу). Для твердих дисків ця операція не має змісту.

У більшості ОС розрізняються, як мінімум, два різних типи драйверів: для символьних і для блокових пристроїв.

Звертаючи до драйвера, ОС указує функцію, що потрібно виконати. Список цих функцій загальний для драйверів різних пристроїв, при цьому кожен драйвер може реалізувати тільки ті функції, що мають сенс для даного пристрою. Найбільш загальними є функції читання даних, запису даних, ініціалізації пристрою (ця функція викликається системою один раз, відразу після завантаження), відкриття і закриття пристрою (використовуються, коли символьний пристрій відкривається як файл). Для блокових пристроїв мають сенс функції форматування, пошуку сектора. Для символьних пристроїв уведення – функція « уведення, щонеруйнує,», тобто перевірки чергового символу без його вилучення з вхідного потоку.

Для того, щоб врахувати вся розмаїтість можливих операцій, у число функцій драйвера вводять таку, як «виконання спеціальних функцій», і тут уже для кожного пристрою визначений свій набір цих спеціальних функцій.

Типовий драйвер пристрою містить, як мінімум, три основних блоки:

72* заголовок драйвера;

73* блок стратегії;

74* блок переривань.

Заголовок містить різну інформацію про даний драйвер і про керований пристрій. Сюди може включатися ім'я пристрою, тип пристрою, число однотипних пристроїв, що обслуговуються одним драйвером, обсяг пам'яті на пристрої і т.п. Заголовок містить також адреси блоку стратегії і блоку переривань.

В обов'язок ***блоку стратегії*** входить прийом заявок на виконання операції, ведення черги заявок (у многозадачных системах, а також при асинхронних операціях, виконання можуть чекати кілька заявок), а також запуск операції і її завершення.

Заявка на виконання операції являє собою стандартний запис, формований системою перед звертанням до драйвера. Заявка містить код необхідної функції драйвера і зведення про адресу даних у пам'яті і на пристрої, про кількість переданих даних. Заявка також містить поле, у яке драйвер повинний буде записати код завершення операції (звичайно 0 – нормально виконана операція, інші значення – коди помилок).

Блок переривань виконує приблизно той алгоритм, що у п. 2.5.1 називався введенням/висновком по перериваннях. Система викликає цей блок, коли одержує сигнал

переривання від пристрою, що обслуговується драйвером. Закінчивши виконання заявки, блок переривань повертає керування блокові стратегії для завершення операції.

Крім трьох основних блоків, у різних ОС драйвери можуть містити, наприклад, блок ініціалізації (він використовується один раз при завантаженні ОС, а потім може бути вивантажений з пам'яті), блок зміни параметрів драйвера й ін.

В останні роки зростаюче ускладнення периферійних пристроїв і самих ОС зробило популярної багаторівневу схему використання драйверів. За цією схемою, крім описаних вище низкоуровневых драйверів апаратури, допускається ще створення высокоуровневых драйверів, що лежать між драйверами апаратури й іншою частиною ОС. Высокоуровневый драйвер не містить блоку переривань, він приймає заявки від системи, перетворить дані тим або іншим способом, а потім викликає низкоуровневый драйвер для роботи з пристроєм. Наприклад, высокоуровневый графічний драйвер може перетворювати команди малювання фігур, заливань, тексту в набір команд конкретної моделі принтера, а зв'язаний з ним драйвер рівнобіжного порту відповідає за передачу цих команд принтерові. Для диска можна реалізувати у виді окремого драйвера алгоритм шифрации даних, що потім передаються звичайному драйверові диска.

2.8. Керування пристроями в MS-DOS

2.8.1. Рівні доступу до пристроїв

Система MS-DOS надає користувачеві можливості доступу до пристроїв на декількох рівнях, що відрізняються ступенем близькості до апаратури. Нижні рівні дозволяють більш повно використовувати тонкі особливості пристроїв, але за это приходиться платити складністю програмування. Верхні рівні більш зручні для рішення стандартних задач уведення/висновку.

Рівні доступу до пристроїв показані на мал. 2- 5.

Уровни доступа к устройствам в MS-DOS

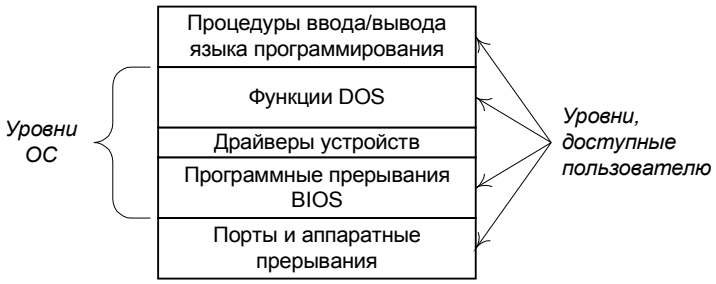


Рис. 2- 5

Самий нижній рівень припускає, що програма користувача працює безпосередньо з портами введення/висновку, а також, якщо це необхідно, частково або цілком бере на себе обробку апаратних переривань, що надходять від пристрою. У принципі, програма користувача може взагалі не використовувати засобу MS-DOS для роботи з пристроями, але при цьому їй прийдеться взяти на себе багато нудної роботи.

У роботі MS-DOS широко використовуються можливості, надані BIOS (Basic Input/Output System, базова система введення/висновку) – набором програмних модулів, записаних у постійній пам'яті (ПЗУ) комп'ютера. BIOS за замовчуванням виконує обробку всіх апаратних переривань, якщо цю роботу не бере на себе DOS або прикладна програма. Крім того, BIOS містить процедури обробки ряду програмних переривань уведення/висновку.

Програмні переривання BIOS являють собою підпрограми, що виконують операції

введення/висновку і керування конкретними пристроями. Для кожного зі стандартних пристроїв зарезервований свій номер переривання, а для вказівки необхідної операції використовується номер функції, що заноситься в один з регістрів процесора перед викликом переривання.

Драйвери пристроїв не викликаються безпосередньо з програми користувача. Вони викликаються з функцій DOS і виконують задану операцію, звичайно використовуючи для цього програмні переривання BIOS, хоча можливо і прямій роботі драйвера з портами й апаратними перериваннями.

Функції DOS, на відміну від переривань BIOS, працюють не з конкретною апаратурою, а з іменованими пристроями. Імена пристроїв задаються в заголовках відповідних драйверів. Наприклад, для DOS клавіатура й екран поєднуються драйвером консольного пристрою CON.

Процедури введення/висновку, використовувані в мовах програмування (наприклад, **`Read`** і **`Write`** у Паскале, **`scanf`** і **`printf`** у C), при компіляції реалізуються як підходящі виклики функцій DOS або, в особливих випадках, програмних переривань BIOS.

2.8.2. Драйвери пристроїв у MS-DOS

Драйвер пристрою в MS-DOS складається з трьох блоків з відомими нам назвами: заголовок драйвера, блок стратегії і блок переривань. При близькому розгляді виявляється, однак, що подібність обмежується в основному назвами блоків.

Заголовок драйвера містить основну інформацію про пристрій: символьний або блоковий пристрій; для символьних пристроїв – ім'я пристрою; для блокових – кількість однотипних пристроїв, що обслуговуються даним драйвером; чи не є даний пристрій системною консоллю, системним годинник або порожнім пристроєм; які спеціальні операції підтримує пристрій.

У заголовку утримуються адреси блоку стратегії і блоку переривань, а також адреса заголовка наступного драйвера в списку. Коли система повинна виконати запит на введення/висновок, воно переглядає список усіх драйверів, поки не знайде пристрій з потрібним ім'ям (якщо задано символьний пристрій) або, для блокового пристрою, не визначить, який по рахунку драйвер відповідає зазначеному номерові (букві) диска. Потім система викликає блок стратегії знайденого драйвера, передаючи йому адреса заявки на виконання операції. Потім викликається блок переривань.

Якщо у файлі конфігурації CONFIG.SYS зазначені імена файлів додаткових (щозавантажуються) драйверів, те ці драйвери містяться в списку перед стандартних системних драйверів. Тому, якщо ім'я пристрою для драйвера, що завантажується, збігається з ім'ям стандартного пристрою MS-DOS, те буде викликаний драйвер, що завантажується, а не стандартний.

Блок стратегії драйвера MS-DOS не робить практично нічого, тільки запам'ятовує адреса заявки. Блок переривань виконує всю роботу по обробці заявки, причому у всіх стандартних драйверів цей блок працює зовсім не по перериваннях, а по опитуванню готовності. Очевидно, розроблювачі першої версії MS-DOS припускали коли-небудь у майбутньому реалізувати нормальну структуру драйвера, так так і не зібралися.

Заявка на виконання операції містить код операції, поле для запису результату операції, номер диска (для блокових пристроїв) і інша дані (наприклад, адреса виведених даних у пам'яті й адресу сектора на диску). Код операції визначає необхідну операцію – наприклад, читання, запис, запис з перевіркою, відкриття або закриття пристрою, перевірка зміни дискети, опитування стану пристрою і т.п. Закінчивши виконання операції, драйвер записує в поле результату, чи була операція виконана успішно або з помилкою, і з який саме (помилка читання, помилка запису, немає папера в принтері, не знайдений сектор на диску, неприпустима операція для даного пристрою і т.п.).

2.8.3. Керування символьними пристроями

Роботу MS-DOS із символьними пристроями цікавіше всего розглянути на прикладі клавіатури. Шлях, що проходять при цьому дані, що вводяться, схематично показаний на мал. 2-6.

Работа с клавиатурой в MS-DOS

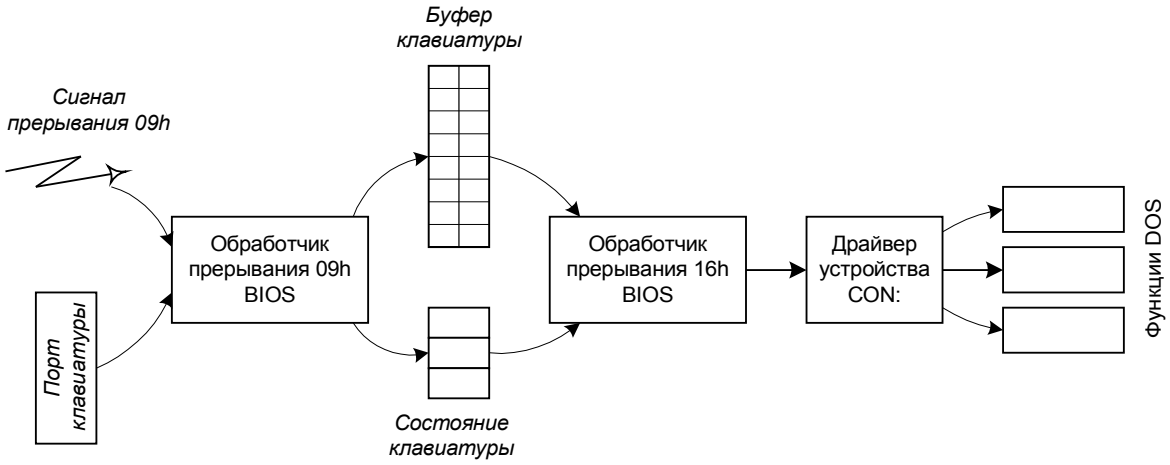


Рис. 2- 6

Коли користувач натискає клавішу, клавіатура переходить у стан готовності і з цього приводу посилає сигнал апаратного переривання **`Int 09h`**. Одночасно в порт, до якого підключена клавіатура, посилається ***скан-код натискання*** клавіші. Цей код являє собою однобайтовое число, що означає порядковий номер натиснутої клавіші. Якщо клавіша довго утримується натиснутої, то через якийсь час починається «автоповтор» – сигнал переривання і скан-код посилаються багаторазово. Нарешті, коли клавіша відпускається, генерується ще одне переривання і посилається ***скан-код відпускання*** клавіші, що відрізняється від коду натискання одиничним значенням старшого біта. Цим практично вичерпуються апаратні події, зв'язані з клавіатурою. Все інше робиться програмно.

Насправді, усі відбувалося саме так зі старою, 83-клавішною клавіатурою комп'ютерів IBM PC XT. Сучасні клавіатури за одне натискання умудряються послати від 1 до 4 скан-кодов підряд. Причини цього пояснювати довго і не дуже цікаво.

Підпрограма BIOS, що обробляє апаратне переривання від клавіатури, повинна, по-перше, запам'ятовувати поточний стан клавіатури: натиснуті чи ні «сдвиговые» клавіші **`Shift`**, **`Ctrl`**, **`Alt`**, включені чи ні режими **`Caps Lock`**, **`Num Lock`**. По-друге, оброблювач повинний з урахуванням цього стану визначити, який символ хотів увести користувач. Та сама клавіша може, наприклад, означати букву ‘Z’ прописну або рядкову, росіянку букву ‘Я’ прописною або рядкову, а також бути частиною комбінацій **`Ctrl+Z`**, **`Alt+Z`**. Відповідний символ буде міститися в буфер клавіатури у виді двох байт: скан-код натиснутої клавіші і ASCII-код символу. Для деяких клавіш і комбінацій, яким не відповідає ніякий ASCII-код (наприклад, **`F1`**, **`Insert`**, **`Ctrl+Home`**, **`Alt+`**буква, →, фірма IBM розробили власний набір «розширених» кодів.

Буфер клавіатури може вмістити до 15 уведених символів, а при переповненні починає огидно пищати.

Програмне переривання **`Int 16h`** також обробляється BIOS’ом. Його призначення – задовольняти запити програм, що звертаються до клавіатури. Найбільше часто використовуються наступні три функції цього переривання.

75* Уведення символу з чеканням. Ця функція повертає значення кодів чергового символу з

буфера клавіатури і видаляє цей символ з буфера. Якщо буфер був порожній, функція виконує активне чекання доти, поки при обробці чергового натискання клавіші в буфері не з'явиться введений символ.

76* Уведення символу без чекання. Він відрізняється тим, що при порожньому буфері не відбувається чекання, а повертається відповідна ознака. Без цієї функції було б неможливо запрограмувати велику частину ігор.

77* Опитування стану клавіатури. Повертає інформацію про поточний стан «сдвиговых» клавіш.

Прикладні програми можуть викликати або переривання **`Int 16h`**, або одну з функцій DOS, призначених для введення символів з консолі. Варто підкреслити, що ці функції працюють не з клавіатурою, а з пристроєм CON (консоллю оператора), через драйвер цього пристрою. Звичайно, практично завжди пристрій CON – це і є клавіатура (плюс ще й екран монітора, що використовується при висновку символів на консоль). Однак теоретично їсти можливість написати нестандартний драйвер пристрою CON, що буде брати символи, що вводяться, наприклад, з вилученого термінала, через модем. Або як консольний пристрій можна використовувати друкарську машинку, як це і робилося раніш, до широкого поширення моніторів.

Набір функцій DOS для введення з консолі досить різноманітний. Однак жодна з цих функцій не використовує особливостей клавіатури як пристрою. Зокрема, функції DOS не знають поняття «стан клавіатури». Зате набір функцій включає введення з «луною-відображенням» уведеного символу на пристрої CON (тобто на екрані) або без відображення, з чеканням або без чекання, введення одного символу або відразу рядка (завершающейся натисканням **`Enter`**), а також уведення з попереднім очищенням буфера (щоб давно завалялися там символи не були випадково введені як відповідь на питання, що задається програмою,).

Коротко розглянемо роботу з іншими символьними пристроями.

Монітор не посилає сигналів переривань і висновок на нього виконується не через порт, а шляхом запису даних в область відеопам'яті. Наявне програмне переривання BIOS практично використовується для переключення відеорежиму (текстове або графічний, число квітів, число крапок на екрані) і для зміни виду і положення курсору текстового режиму. Висновок символів через BIOS досить повільний і використовується в основному для невеликих текстових повідомлень. У графічних режимах BIOS працює неприйнятно повільно, тому що він за кожен виклик переривання може вивести тільки одну крапку. Серйозні програми працюють з відеопам'яттю прямо. Функції DOS для висновку на консоль не мають навіть права використовувати таку специфіку монітора, як керування курсором і вибір кольору, адже вони працюють із пристроєм CON, що може виявитися і друкарською машинкою.

Для принтера BIOS пропонує можливість перевірки стану пристрою (зокрема, чи знаходиться принтер у стані готовності) і висновку одного символу з опитуванням готовності. У той же час, DOS, крім убогої функції висновку одного символу з чеканням на пристрій PRN, дозволяє використовувати системну програму **`PRINT`**, що уміє виводити файли у фоновому режимі, не заважаючи одночасно виконувати інші програми. Ця програма працює з апаратними перериваннями в обхід BIOS.

Про роботу з послідовним портом можна сказати майже ту ж саме, що про роботу з принтером. Відмінність у тім, що порт може працювати і на введення даних. Один із самих вірних способів «підвісити» систему навічно – запросити введення символу з COM-порту з чеканням через BIOS або DOS, якщо на вхід порту не надходять ніякі дані. Практично завжди робота з COM-портом ведеться за допомогою нестандартних драйверів, що працюють по перериваннях.

Миша взагалі не є стандартним пристроєм для комп'ютерів – нащадків IBM PC. BIOS нічого не знає про існування мишей. Драйвери миші мало схожі на інші драйвери MS-DOS. Вони дозволяють опитувати поточне положення миші і стан кнопок, однак більш зручної для додатків є можливість задати за допомогою драйвера власну процедуру обробки переривань від миші. Ця процедура повинна визначати, у якому місці екрана був курсор при натисканні кнопки миші, який елемент керування (кнопка, пункт меню, смуга прокручування і т.п.) знаходиться в цій крапці, і тільки потім – яка дія варто почати при щиглику миші на цьому елементі. Занадто багато чорнової роботи, що більш сучасні ОС (наприклад, Windows) беруть на себе.

2.8.4. Керування блоковими пристроями

2.8.4.1. Структура диска

Основним видом блокових пристроїв є магнітні й інші диски, тому почнемо з розгляду структури диска (мал. 2- 7).

Структура магнитных дисков

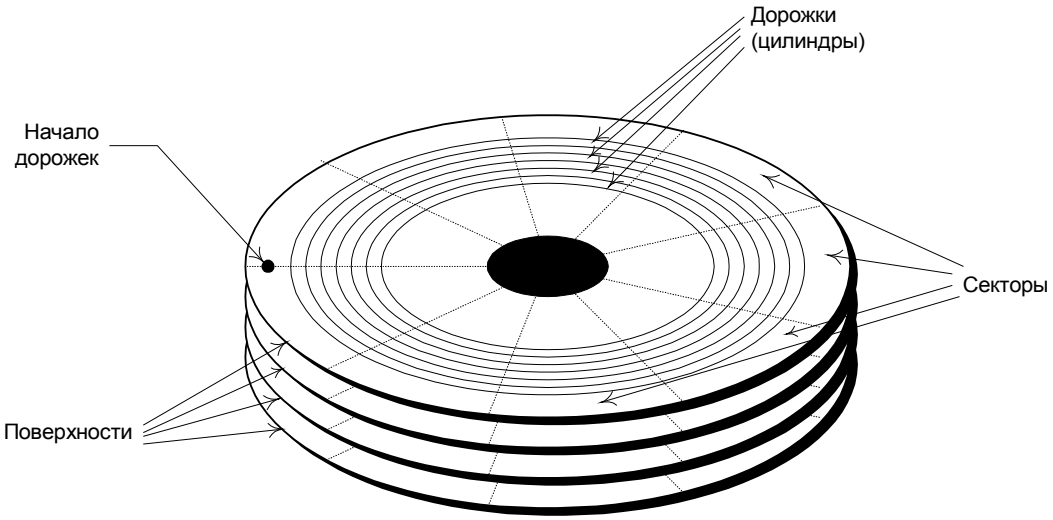


Рис. 2- 7

Поверхня нового магнітного диска покрита однорідним шаром магнітного матеріалу. У дискети використовується або одна поверхня, або (частіше) обидві поверхні. Число поверхонь твердого дискового тому визначається кількістю дисків, з яких зібраний тім.

Першою операцією, що повинна бути пророблена з диском, є ***низкоуровневое форматування***. Воно полягає в розмітці поверхні на ***доріжки*** магнітного запису, розділені на ***сектори***. Відстань між доріжками визначається кроком переміщення голівок читання/запису, а розбивка на сектори виконується програмно, шляхом запису даних на доріжки в моменти, розраховані на підставі відомої швидкості обертання диска. Для всіх операцій з диском, крім низкоуровневого форматування, сектор є мінімальною одиницею читання або запису даних.

Сукупність доріжок однакового радіуса на всіх поверхнях диска називається ***циліндром***.

Структура сектора показана на мал. 2- 8.

Структура сектора диска

Рис. 2- 8

Сектор складається з заголовка, блоку даних і контрольної інформації (контрольної суми), що служить для перевірки правильності зчитування сектора.

Заголовок сектора містить ***фізична адреса*** сектора і його розмір. Фізична адреса складається з трьох чисел: номер циліндра, номер поверхні і номер сектора на доріжці. Найперший сектор диска має адреса (0, 0, 1). Розмір сектора на IBM-сумісних комп'ютерах завжди дорівнює 512 байт.

Один час був модно використовувати нестандартний розмір окремих секторів на дискеті (наприклад, 1024 байта) для утруднення несанкціонованого копіювання. Дискету, що містить нестандартний сектор, могла правильно прочитати тільки програма, який було точно відомо про наявність такого сектора.

Між секторами й усередині секторів маються проміжки, що використовуються апаратурою при пошуку заданого сектора.

Нумерація секторів не обов'язково ведеться в порядку їхнього розміщення на доріжці. Якщо швидкість наявної апаратури недостатня для того, щоб устигнути прочитати і передати в пам'ять дані з усієї доріжки за час одного обороту диска, то система при форматуванні нумерує сектори «через один» або навіть «через два». Наприклад, при 9 секторах на доріжці вони можуть бути пронумеровані «через один» у такому порядку: 1, 6, 2, 7, 3, 8, 4, 9, 5. Після читання сектора 1 у контролера диска їсти час передати прочитані дані, поки до голівки читання не підійде сектор 2. У результаті вся доріжка може бути прочитана за два обороти диска.

2.8.4.2. Розділи і логічні томи

Загальна структура дискет і твердих дисків розрізняються між собою. Ці структури показані на мал. 2- 9.

Структура дисков

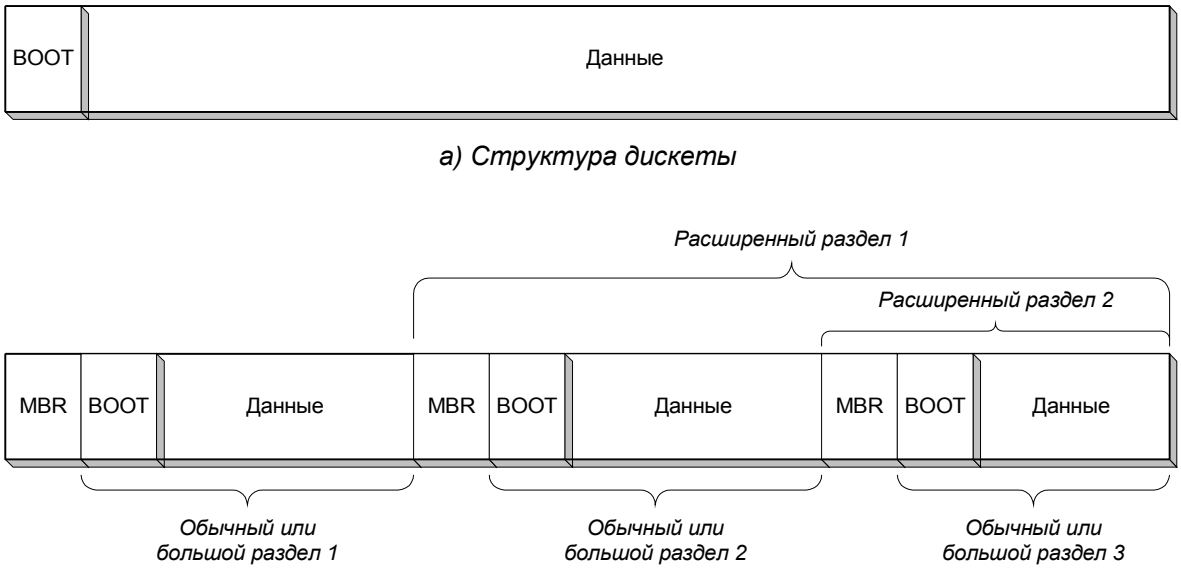


Рис. 2- 9

Початковий сектор дискети (мал. 2- 9, а) прийнято називати ***BOOT-сектором***. Він містить кількісні дані про дискету (розмір секторів, кількість секторів на кожній доріжці і на всій дискеті, число поверхонь і т.п.), мітку (назва) і серійний номер дискети, а також дані про файлову систему. Крім того, якщо дискета містить системні файли ОС, то в BOOT-секторі знаходиться також невелика програма початкового завантаження, що зчитує один сектор ОС і передає йому керування для продовження завантаження. Всі інші сектори дискети можуть використовуватися ОС для збереження її файлів і інших даних. Загальна кількість секторів на дискеті не може перевищувати 2^{16} (насправді, їх значно менше).

Скажіть швидко, скільки приблизно секторів містить стандартна тридюймова дискета?

Для твердого диска (мал. 2- 9, б) початковий сектор називається MBR (Master Boot Record, головний завантажувальний запис). Він теж може містити програму початкового завантаження, але, крім того, містить ***таблицю розділів*** (partition table), що описує розбивку твердого диска на ***розділи***.

Таблиця може містити від 1 до 4 записів про розділи. Кожен запис містить тип розділу, число секторів у ньому, фізичні адреси початку і кінця роздягнула.

Можливі наступні типи розділів.

78* ***Звичайний*** розділ. Його структура точно така ж, як у дискети, тобто такий розділ починається з BOOT-сектора, а загальне число секторів не перевищує 2^{16}. Таким чином, загальний розмір роздягнула не може перевищувати 32 Мб.

79* ***Великий*** розділ. Він відрізняється від звичайного тем, що число секторів може досягати 2^{32}. Це дозволяє описувати великі розділи розміром до 2048 Гб.

80* ***Розширений*** розділ. Його структура аналогічна структурі усього твердого диска, тобто початковий сектор роздягнула – не BOOT, а MBR-сектор. Аналогія не зовсім повна, оскільки таблиця розділів у MBR розширеного розділу може містити не більш двох записів, причому перша з них повинна описувати або звичайний, або великий розділ, а другий запис, якщо вона мається, описує ще один розширений розділ.

81* Розділи інших ОС (наприклад, UNIX).

Звичайні і великі розділи називаються також ***логічними томами*** або ***логічними дисками***, на відміну від фізичних дисків. Звичайна буквена нумерація дисків A, B, C, D і т.д. відноситься саме до логічних томів. Для дискет поняття фізичний і логічний томи збігаються.

Споконвічно MS-DOS підтримувала тільки звичайні розділи на твердому диску. У 80-і роки здавалося, що 32 Мб – це дуже великий обсяг диска. Коли з'явилися диски обсягом у кілька сотень мегабайт, була придумана матрешечная структура розширених розділів, що дозволило на одному фізичному томі розмістити скільки завгодно логічних томів по 32 Мб. Потім були реалізовані великі розділи, що зажадало від розроблювачів MS-DOS внести істотні зміни в програмний інтерфейс і реалізацію засобів роботи з дисками. Після цього використання розширених розділів стало необов'язковим, якщо користувачеві досить мати не більш чотирьох логічних томів на одному фізичному диску.

2.8.4.3. Засобу доступу до дисків

На рівні архітектури системи контролер диска представлений декількома регістрами, доступними через порти комп'ютера. У ці порти перед виконанням операцій заносяться такі дані, як номер диска, кількість секторів, що беруть участь в операції і фізичну адресу початкового сектора, записувані дані. Один з портів служить для посилки команди на виконання операції (читання, запис, запис з перевіркою, пошук циліндра, форматування доріжки й ін.). Коли операція довершена, контролер посилає сигнал переривання.

Прикладні програми можуть працювати з дисками або засобами BIOS (програмне переривання **`int 13h`**), або засобами DOS (програмні переривання **`int 25h`** – читання і **`int 26h`** – запис). При цьому BIOS працює тільки з фізичними дисками, використовуючи фізичну нумерацію секторів, у той час як MS-DOS працює тільки з логічними томами, а сектори при цьому нумеруються числами від 0 (що відповідає BOOT-секторові) до максимального номера сектора на томі. На практиці те й інше використовується досить рідко. Більшість прикладних програм не працюють з дисками як із пристроями, замість цього робота ведеться на рівні файлового введення/висновку, а відображення файлів на сектори диска є обов'язком ОС.

Для підвищення ефективності роботи з дисками в MS-DOS використовуються кэширование дисків і випереджальне читання, описані в пп. 2.6.6 і 2.6.7. Дисковий кэш у MS-DOS улаштований простіше, ніж у UNIX. По-перше, в однозадачной системі немає необхідності в списку вільних блоків (коли процес звертається до диска, немає інших процесів, що могли б у цей час працювати з буферами). По-друге, розмір кэша рідко перевищує 30 – 40 буферів, тому не потрібні і хеш-цепочки, усі буфери об'єднані в єдиному LRU-списку.

Чому в MS-DOS не потрібний такий великий кэш, як у UNIX?

2.9. Керування пристроями в Windows

2.9.1.1. Драйвери пристроїв у Windows

Оскільки Windows – многозадачная система, вона виключає для прикладних програм такі вільності, як пряме звертання до портів уведення/висновку або обробка апаратних переривань. Взаємодія з апаратурою на низькому рівні може виконуватися тільки системними програмами, що працюють у привілейованому режимі. Основну роль тут грають драйвери пристроїв.

У Windows використовується багаторівнева структура драйверів, у якій высокоуровневые драйвери можуть відігравати роль фільтрів, що виконують спеціальну обробку даних, отриманих від драйвера низького рівня або переданих такому драйверові. Як приклад можна привести відділення драйвера, що керує шиною, від драйверів конкретних пристроїв, підключених до

шини. Ще один приклад – драйвер, що виконує шифрацию/дешифрацию даних при роботі з файловою системою NTFS. Структура драйверів усіх рівнів підлегла єдиним стандартам, відомим як WDM (Windows Diver Model), однак высокоуровневые драйвери, на відміну від низкоуровневых, не займаються обробкою апаратних переривань.

Як не дивно, у Windows NT низкоуровневые драйвери – це ще не самий нижній рівень керування пристроями. Ще ближче до апаратури лежить так називаний рівень HAL (Hardware Abstractions Level, рівень апаратних абстракцій). Його роль – сховати від інших модулів ОС, у тому числі і від драйверів, деякі деталі роботи з апаратурою, що залежать від конкретних шин, типу материнської плати, способу підключення. Наприклад, HAL надає драйверам можливість звертатися до регістрів пристроїв по їхніх логічних номерах, не знаючи при цьому, чи підключений регістр до порту процесора або відображений на пам'ять.

Незважаючи на стандартизацію структури, можна виділити кілька спеціальних типів драйверів, що відрізняються функціональним призначенням.

82* Драйвери GDI (Graphic Device Interface) являють собою высокоуровневые драйвери графічних пристроїв (моніторів, принтерів, плоттеров). Ці драйвери виконують трансляцію графічних викликів Windows (таких, як «провести лінію», «залити область», «видати текст», «вибрати поточний шрифт, що текет перо, що текет заливання») у команди, що виконують відповідні дії на конкретному пристрої. Видача цих команд на пристрій виконується вже іншим, низкоуровневым драйвером. Завдяки наявності драйверів GDI та сама програма може видавати графічне зображення на різні пристрої. Яскравий приклад цього – наявний у різних редакторах режим попереднього перегляду, що відображає сторінки на екрані точно в тім виді, як вони будуть надруковані.

83* Драйвери клавіатури і миші, крім стандартних для драйвера операцій, виконують додаткове навантаження. Вони генерують повідомлення про події на відповідному пристрої (натискання і відпускання клавіші, переміщення миші, натискання і відпускання кнопок миші) і поміщають них у системну чергу повідомлень. Потім система переправляє кожне повідомлення процесові, якому воно було призначено, для подальшої обробки.

84* Драйвери виртуализации пристроїв (VxD-драйвери) служать для того, щоб розділяти пристрою між процесами, створюючи ілюзію, що процес монопольно володіє пристроєм. Насправді драйвер організує черга заявок від процесів, переключає пристрій у потрібен для чергового процесу режим і т.п. Прикладом може служити драйвер виртуализации монітора. Консольний додаток (наприклад, програма MS-DOS) працює з всім екраном у текстовому режимі. Але якщо такий додаток запущений у вікні Windows, те VxD-драйвер імітує текстовий режим у графіку. Для цього драйвер повинний перехоплювати спроби програми звернутися прямо до адрес відеопам'яті і перетворювати координати знакомест текстового режиму в координати відповідних позицій у вікні.

2.9.1.2. Доступ до пристроїв

У більшості випадків програми не працюють безпосередньо з пристроями. Замість цього для виконання необхідних операцій використовуються API-функції більш високого рівня, а звертання до пристроїв виконуються системою в міру потреби. Наприклад, файлові функції звертаються в кінцевому рахунку до дискових пристроїв, а функції GDI працюють з монітором або з принтером, у залежності від зазначеного контексту пристрою.

У ряді випадків програміст усе-таки може віддати перевагу безпосередній роботі з пристроєм. Щоб одержати доступ до пристрою, програма повинна відкрити цей пристрій викликом тієї ж API-функції **`CreateFile`**, що використовується і для відкриття файлів. У даному випадку замість імені файлу варто вказати ім'я драйвера пристрою, що відкривається.

Для дискових пристроїв можна замість імені драйвера вказати ім'я самого пристрою. Наприклад, ім'я «**\\.\C:**» означає логічний диск C, а ім'я «**\\.\PHYSICALDRIVE0**» – перший фізичний диск комп'ютера.

Відкривши пристрій, програма може або читати або записувати дані, використовуючи функції файлового введення/висновку, або видавати команди керування пристроєм за допомогою функції **DeviceIoControl**. За допомогою цих команд можна, наприклад, отформатировать диск і розбити його на розділи, завантажити або витягти CD-ROM диск, змінити деякі параметри роботи модему і т.п..

2.10. Керування пристроями в UNIX

2.10.1. Драйвери пристроїв у UNIX

Драйвери в ОС UNIX досить точно відповідають стандартній схемі драйвера, приведеної в п. 2.7. Проте, через істотні розходження в роботі із символьними і з блоковими пристроями, у UNIX розрізняються два основних типи драйверів: символьні і блокові.

Для символьних пристроїв використовуються тільки символьні драйвери. Для кожного блокового пристрою звичайно мається два різних драйвери: блоковий і символьний. Блоковий драйвер дозволяє виконувати операції тільки з цілим числом блоків, як і покладено працювати з блоковими пристроями. Символьний драйвер блокового пристрою є більш высокоуровневой програмою, що імітує виконання операцій читання і записи довільної кількості байт, насправді використовуючи звертання до блокового драйвера.

Крім драйверів реальних фізичних пристроїв, система може включати драйвери «псевдоустройств». Прикладом може служити драйвер, що забезпечує звертання програм до вмісту системної пам'яті.

При завантаженні системи формуються дві таблиці, для символьних і для блокових драйверів. Рядка таблиці відповідають конкретним драйверам, а стовпці – функціям, що повинний уміти виконувати драйвер, так що в осередках таблиці утримуються адреси, по яких викликаються функції драйвера. Набір функцій для символьних і для блокових драйверів злегка відрізняється, тому використовуються дві різних таблиці.

До найбільш важливих функцій драйвера відносяться наступні.

85* Відкриття пристрою. Як мінімум, при цьому збільшується лічильник поточних звертань до пристрою, що дозволяє ставити звертання в чергу, якщо пристрій зайнятий. Деякі пристрої при відкритті можуть виконувати ще якісь початкові дії.

86* Закриття пристрою – операція, протилежна відкриттю.

87* Обробка переривання – виконує введення або висновок чергової порції даних, коли пристрій переходить у стан готовності.

88* Опитування пристрою – ця функція виконується для тих пристроїв, що не генерують переривань, або якщо при розробці драйвера або вирішено не використовувати переривання від пристроїв. Опитування виконується не постійно, а з деяким періодом, по перериванню від таймера.

89* Читання даних із пристрою.

90* Запис даних на пристрій.

91* Виклик стратегії. Це спосіб виконання операцій уведення/висновку, характерний для блокових пристроїв. При цьому запит може бути поставлений у чергу. Запит у ряді випадків може бути вдоволений шляхом звертання до дискового кэшу (див. п. 2.6.6), без виконання читання або запису на пристрій.

92* Виконання спеціальних функцій. Набір цих функцій залежить від конкретного пристрою. Це може бути, наприклад, опитування або установка поточного режиму роботи пристрою, форматування доріжок диска, перемотування стрічки і т.п.

2.10.2. Пристрій як спеціальний файл

Цікавою відмінною рисою UNIX є те, що для роботи з периферійними пристроями прикладні програми можуть і повинні використовувати ті ж засоби, що для роботи з файлами. Узагалі, пристрою в UNIX представлені як ***спеціальні файли***, уписані в каталог файлової системи нарівні зі звичайними файлами. Кожному драйверові пристрою відповідає окремий спеціальний файл, символьній або блоковий, у залежності від типу драйвера. Як правило, усі спеціальні файли розміщаються в каталозі **`/dev`**. Щоб почати роботу з пристроєм, програма повинна викликати функцію відкриття файлу, указавши їй ім'я спеціального файлу. При цьому відбувається звертання до функції відкриття з драйвера відповідного пристрою.

З кожним спеціальним файлом зв'язані два числа, називані старшим і молодшим номерами пристрою. Старший номер визначає номер рядка в таблиці символьних або блокових драйверів. Молодший номер передається драйверові як додатковий параметр. Він може означати, наприклад, номер конкретного дискового пристрою.

3. КЕРУВАННЯ ДАНИМИ

3.1. Основні задачі керування даними

Стародавній термін «керування даними» у даний час завжди розуміється як керування файлами.

Файл є набір даних, що зберігається на периферійному пристрої і доступний по імені. При цьому конкретне розташування даних на пристрої не цікавить користувача і цілком передоручається системі. До винаходу файлів користувач повинний був звертатися до своїх даних, указуючи їхньої адреси на диску або на магнітній стрічці.

Поняття «***файлова система***» означає стандартизовану сукупність структур даних, алгоритмів і програм, що забезпечують збереження файлів і виконання операцій з ними. Могутня сучасна ОС звичайно підтримує можливість використання декількох різних файлових систем. І навпаки, та сама файлова система може підтримуватися різними ОС.

Серед задач, розв'язуваних підсистемою керування даними, можна назвати наступні:

93* виконання операцій створення, видалення, перейменування, пошуку файлів, читання і записи даних у файли, а також ряду допоміжних операцій;

94* забезпечення ефективного використання дискового простору і високої швидкості доступу до даних;

95* забезпечення надійності збереження даних і їхнього відновлення у випадку збоїв;

96* захист даних користувача від несанкціонованого доступу;

97* керування одночасним спільним використанням даних з боку декількох процесів.

3.2. Характеристики файлів і архітектура файлових систем

З кожним файлом зв'язаний набір ***атрибутів*** (***характеристик***), тобто набір зведень про файл. Склад атрибутів може сильно розрізнятися для різних файлових систем. Приведемо зразковий список можливих атрибутів, не прив'язуючи до якої-небудь конкретної системи.

98* ***Ім'я файлу***. У старих ОС довжина імені була жорстко обмежена 6 – 8 символами з метою економії місця для збереження імені і прискорення роботи. В даний час максимальна довжина імені складає звичайно близько 250 символів, що дозволяє при бажанні включити в ім'я файлу докладний опис його вмісту.

99* ***Розширення імені***. За традицією, так прийнято називати праву частину імені, відділену крапкою. У MS-DOS, як і в деяких більш ранніх системах, цей атрибут не є частиною імені, він зберігається окремо й обмежується по довжині 3 символами. Однак зараз узяв гору підхід, прийнятий у UNIX, де розширення – це чисте умовно виділювана частина імені після останньої крапки. Розширення звичайне вказує тип даних у файлі.

100* ***Тип файлу***. Деякі ОС виділяють трохи істотно різних типів файлів, наприклад, символьні і двоичные, файли даних і файли програм і т.п. Нижче будуть розглянуті типи файлів, що розрізняються UNIX.

101* ***Розмір файлу***. Звичайно вказується в байтах, хоча раніш часто задавався в блоках.

102* ***Тимчасові штампи***. Під цим терміном розуміються різні оцінки дати і, може бути, часу дня. Найважливішим з тимчасових штампів є час останньої модифікації, що дозволяє визначити найбільш свіжу версію файлу. Корисними можуть бути також час останнього доступу (тобто відкриття файлу), час останньої модифікації атрибутів.

103* ***Номер версії***. У деяких ОС при всякій зміні файлу створювалася його нова версія, причому система могла зберігати або усі версії, або тільки трохи останніх. Це давало немаловажну перевагу – можливість повернутися до старої версії файлу, якщо зміни виявилися невдалими. Проте, цей атрибут не прищепився через велику надлишкову витрату дискової пам'яті. При необхідності розроблювачі можуть використовувати спеціальні програмні системи керування проектами, що забезпечують у тому числі і збереження старих версій файлів.

104* ***Власник файлу***. Цей атрибут необхідний у многопользовательских системах для організації захисту даних. Як правило, власником є користувач, що створив файл. Іноді, крім індивідуального власника, указується ще і група користувачів як колективний власник файлу.

105* ***Атрибути захисту***. Вони вказують, які саме права доступу до файлу мають різні користувачі, у тому числі і власник файлу.

106* ***Тип доступу***. У деяких ОС (наприклад, у OS/360) для кожного файлу повинний був зберігатися припустимий тип доступу: послідовний, довільний або один з індексних типів, що забезпечують швидкий пошук даних у файлі. В даний час більш розповсюджений підхід, при якому для усіх файлів підтримуються ті самі типи доступу (послідовне і довільний), а прискорення пошуку повинне забезпечуватися, наприклад, системою керування базами даних.

107* ***Розмір запису***. Якщо ця величина зазначена, то адресація потрібних даних виконується за допомогою номера запису. Інший підхід полягає в тім, що дані адресуються їхнім зсувом (у байтах) від початку файлу, а розбивка файлу на записі покладається на прикладні програми, що працюють з файлом.

108* ***Прапори*** (***бітові атрибути***). Їхня розмаїтість обмежується лише фантазією розроблювачів системи, але найбільш розповсюджений і важливим є прапор «тільки для читання» (read only), що захищає файл від випадкової зміни або видалення. У залежності від можливостей конкретної файлової системи, файл може бути відзначений як «стиснутий», «шифрований» і т.п.

109* ***Дані про розміщення файлу на диску***. Користувач, як правило, не знає і не хоче нічого знати про розміщення файлу (саме для цього й існує поняття файлу). Для системи ці дані

необхідні, щоб знайти файл.

Запису, у яких утримуються атрибути кожного файлу, зібрані в ***каталоги*** (вони ж папки, директорії). У ранніх ОС (і навіть у першій версії MS-DOS) на кожнім дисковому томі мався єдиний каталог, що містить повний список усіх файлів цього тому. Таке рішення було цілком природним, поки кількість файлів не перевищувало двох – трьох десятків. Однак при збільшенні обсягу дисків і, як наслідок, числа файлів на них такий одноуровневый каталог ставав усе менш зручним. У деяких ОС використовувалася дворівнева організація каталогів. При цьому головний каталог містив список каталогів другого рівня, закріплених за окремими користувачами або проектами. Однак пізніше стала загальноприйнятої ***ієрархічна структура каталогів***, при якій кожен каталог може, крім файлів, містити вкладені підкаталоги, причому глибина вкладення не обмежується.

Усі службові дані, що зберігаються у файловій системі, що описують атрибути і розміщення файлів, структуру каталогів, загальну структуру дискового тому і т.п., прийнято ***називати*** метаданными, на відміну від «просто даних», що зберігаються у файлах.

Крім пристроїв довільного доступу (дисків), файли можуть зберігатися і на таких пристроях послідовного доступу, як магнітні стрічки. Однак для стрічок ведення каталогів важко і користь від них невелика. Як правило, ім'я та інші атрибути файлу записуються на стрічку безпосередньо перед даними цього файлу.

3.3. Розміщення файлів

Область даних диска, відведену для збереження файлів, можна представити як лінійну послідовність адресуемых блоків (секторів). Розміщаючи файли в цій області, ОС повинна відвести для кожного файлу необхідна кількість блоків і зберегти інформацію про те, у яких саме блоках розміщений даний файл. Існують два основних способи використання дискового простору для розміщення файлів.

110* ***Безперервне розміщення*** характеризується тим, що каждый файл займає безперервну послідовність блоків.

111* ***Сегментированное розміщення*** означає, що файли можуть розміщатися «по шматочках», тобто один файл може займати кілька несуміжних сегментів різної довжини. Обидва способи розміщення показані на мал. 3- 1.

Способы размещения файлов

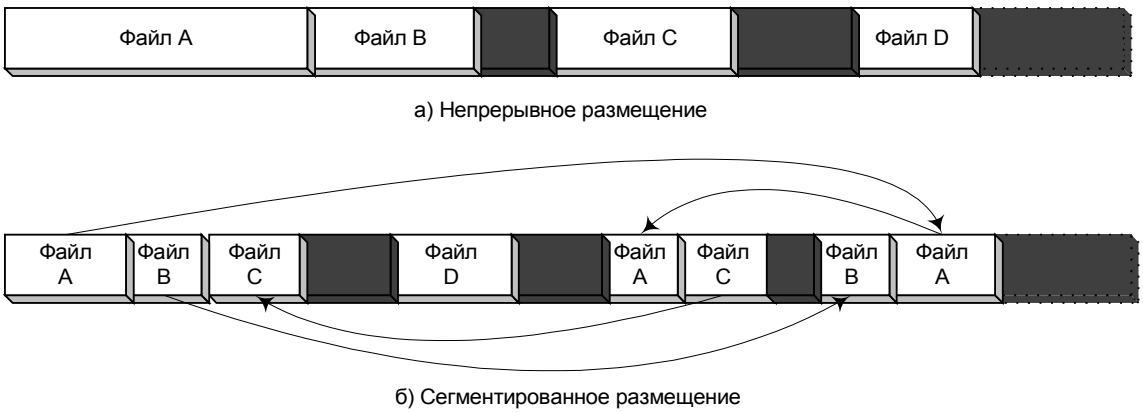


Рис. 3- 1

Безперервне розміщення має два серйозних достоїнства.

112* Інформація про розміщення файлу дуже проста і займає мало місця. Фактично досить зберігати два числа: номер початкового блоку файлу і число займаних блоків (або розмір файлу в байтах, по якому легко обчислити число блоків).

113* Доступ до будь-якої позиції у файлі виконується швидко, оскільки, знаючи зсув від початку файлу, легко можна обчислити номер необхідного блоку і прочитати відразу цей блок, не читаючи попередні блоки.

На жаль, недоліки безперервного розподілу ще більш вагомі.

114* При створенні файлу потрібно заздалегідь знати його розмір, щоб знайти і зарезервувати на диску область достатньої величини. Наступне можливе збільшення файлу досить утруднено, тому що після кінця файлу може не виявитися досить вільного місця. Фактично замість збільшення файлу звичайно приходиться заново створювати файл більшого розміру в іншому місці, переписувати в нього дані і видаляти старий файл. Але таке рішення вимагає багато часу на читання і запис даних і, крім того, знижує надійність збереження даних, оскільки помилка при читанні або записі набагато більш ймовірний, чим псування даних, «спокійно лежачих» на диску.

115* У ході звичайної експлуатації файлової системи, після багаторазового створення і видалення файлів різної довжини, вільний простір на диску виявляється розбитим на невеликі шматочки. Сумарний обсяг вільного місця на диску може бути досить великим, але створити файл пристойного розміру не вдається, для нього немає безперервної області потрібної довжини. Це явище зветься ***фрагментації диска***. Для боротьби з ним приходиться використовувати спеціальну процедуру дефрагментации, що переміщає усі файли, розміщаючи їхній упритул друг до друга від початку області даних диска. Але така процедура вимагає багато часу, знижує, як сказано вище, надійність і збільшує проблеми у випадку, якщо пізніше буде потрібно збільшити файл.

Сегментированное розміщення позбавлене першого з недоліків безперервного: при створенні файлу йому звичайно взагалі не виділяють пам'ять, а потім, у міру зростання розміру файлу, йому можуть бути виділені будь-які вільні сегменти на диску, незалежно від їхньої довжини.

Не так просто з фрагментацією. Звичайно, на відміну від безперервного розміщення, при сегментированном ніяка фрагментація не перешкодить системі використовувати всі блоки, що маються на диску. Однак послідовне читання із сегментированного файлу може виконуватися істотно повільніше за рахунок необхідності переходити від сегмента до сегмента. Уповільнення особливе помітно, якщо файл виявився розкиданий маленькими шматочками по декількох циліндрах диска. У результаті, час від часу доцільно виконувати дефрагментацию диска, щоб підвищити швидкість доступу до даних. При сегментированном розміщенні дефрагментация означає не тільки об'єднання усіх вільних ділянок диска, але і, головним чином, об'єднання сегментів кожного файлу. Ця процедура виконується значно складніше, ніж дефрагментация при безперервному розміщенні.

Чи можете ви запропонувати гарний алгоритм дефрагментации? Врахуйте, що він повинний ефективно працювати, навіть якщо на диску залишилося усього кілька вільних блоків.

Недоліком сегментированного розміщення є те, що інформація про розміщення файлу в цьому випадку набагато складніше, ніж для безперервного випадку і, що найбільше неприємно, обсяг цієї інформації перемінний: чим більше число сегментів займає файл, тим більше потрібно інформації, тому що треба перелічити всі сегменти. Мається майже стільки ж способів рішення цієї проблеми, скільки взагалі придумано різних файлових систем.

Щоб зменшити вплив сегментації на швидкість доступу до даних файлу, в ОС, що використовують сегментированное розміщення, застосовуються різні алгоритми вибору місця

для файлу. Їхньою метою є розмістити файл по можливості в одному сегменті, і тільки в крайньому випадку розбивати файл на кілька сегментів.

У сучасних ОС для файлових систем на магнітних дисках практично завжди використовують сегментированное розміщення. Інша справа файлові системи на дисках, призначених тільки для читання (наприклад, CD ROM). Неважко зрозуміти, що в цьому випадку недоліки безперервного розміщення не мають ніякого значення, а його достоїнства зберігаються.

Ще одною важливою характеристикою розміщення файлів є ступінь його «дробности». Дотепер ми припускали, що файл може займати будь-як ціле число блоків, а під блоком фактично розуміли сектор диска. Проблема в тім, що для дисків великого обсягу число блоків може бути занадто великим. Допустимо, у деякій файловій системі розмір блоку дорівнює 512 байт, а для збереження номерів блоків файлу використовуються 16-розрядні числа. У цьому випадку розмір області даних диска не зможе перевищити $512 * 2^{16}$ = 32 Мб, що нині смішно. Звичайно, можна перейти до використання 32-розрядних номерів блоків, але тоді сумарний розмір інформації про розміщення усіх файлів на диску стає надто великим. Звичайний вихід з цього утруднення полягає в тім, що мінімальною одиницею розміщення файлів вважають ***кластер*** (називаний у деяких системах ***блоком*** або ***логічним блоком***), що приймається рівним 2^k секторів, тобто, наприклад, 1, 2, 4, 8, 16, 32 сектора, рідко більше. Кожному файлові приділяється ціле число кластерів, і в інформації про розміщення файлу зберігаються номери кластерів, а не секторів. Збільшення розміру кластерів дозволяє скоротити кількість даних про розміщення файлів «і в довжину й у ширину»: по-перше, для кожного файлу потрібно зберігати інформацію про менше число кластерів, а по-друге, зменшується число двоичных розрядів, використовуваних для завдання номера кластера (або при тій же розрядності можна використовувати більший диск). Так, при кластері розміром 32 сектора і 16-розрядних номерів можна адресувати до 1 Гб дискової пам'яті.

Використання великих кластерів має свою погану сторону. Оскільки розмір файлу можна вважати випадковою величиною (принаймні, цей розмір ніяк не зв'язаний з розміром кластера), те можна приблизно вважати, що в середньому половина останнього кластера кожного файлу залишається незайнятої. Це явище іноді називають ***внутрішньою фрагментацією*** (на відміну від описаної вище фрагментації вільного простору диска, що називають також ***зовнішньою*** фрагментацією). Крім того, якщо хоча б один із секторів, що входять у кластер, відзначений як дефектний, те і весь кластер вважається дефектним, тобто не може бути використаний. Очевидно, що при збільшенні розміру кластера зростає і число невикористовуваних секторів диска.

Оптимальний розмір кластера або обчислюється автоматично при форматуванні диска, або задається вручну.

Для нормальної роботи файлової системи потрібно, щоб, крім інформації про розміщення файлів, система зберігала в зручному для використання виді інформацію про наявні вільні кластери диска. Ця інформація необхідна при створенні нових або збільшенні існуючих файлів. Використовуються різні способи представлення інформації про вільне місце, деякі з них перераховані нижче.

116* Можна зберігати усі вільні кластери як зв'язаний лінійний список, тобто на початку кожного вільного кластера зберігати номер наступних за списком. Недолік такого способу в тім, що утрудняється пошук вільного безперервного фрагмента потрібного розміру, тому складніше оптимизировать розміщення файлів.

117* Названий недолік можна перебороти, якщо зберігати список не з окремих кластерів, а з безперервних вільних фрагментів диска. Правда, працювати з таким списком трохи складніше.

118* У системах з безперервним розміщенням часто кожен безперервний фрагмент диска описують так само, як файл, але відзначають його прапорцем «вільний».

119* Зручний і простий спосіб полягає у використанні ***бітової карти*** (bitmap) вільних кластерів. Вона являє собою масив, що містить по одному біті на кожен кластер, причому значення 1 означає «кластер зайнятий», а 0 – «кластер вільний». Для пошуку вільного безперервного фрагмента потрібного розміру система повинна буде переглянути весь масив.

3.4. Захист даних

У многопользовательских ОС першорядного значення набуває задача захисту даних користувача від випадкового або навмисного доступу з боку інших користувачів. Питання захисту даних і стандартизації вимог до безпеки ОС заслуговують вивчення в окремому курсі, тому тут вони будуть розглянуті дуже коротко.

Як відзначалося в п. 1.6, для реалізації многопользовательской захисту даних необхідна наявність апаратних засобів, таких як привілейований режим роботи процесора. У противному випадку будь-яка чисто програмна система захисту міг би бути порушена за допомогою досить витонченої програми злому.

Для будь-якої системи захисту характерне наявність, принаймні, трьох компонентів.

120* Список користувачів системи, що містить імена, паролі і привілеї, привласнені користувачам.

121* Наявність атрибутів захисту у файлів і інших об'єктів, що захищаються. Ці атрибути вказують, хто з користувачів має право доступу до даного об'єкта і які саме операції йому дозволені.

122* Процедура ***аутентификации користувача***, тобто встановлення його особистості при вході в систему. Такі процедури найчастіше засновані на введенні пароля, хоча можуть використовуватися і більш екзотичні засоби (відбитки пальців, спеціальні картки і т.п.).

Крім окремих користувачів, визначеними правами доступу до об'єктів можуть володіти ***групи користувачів***. Поняття групи полегшує адміністрування прав доступу. Замість того, щоб індивідуально вказувати набір прав для кожного користувача, досить зарахувати його в одну або кілька груп, права яких визначені заздалегідь.

Нормальне обслуговування системи захисту неможливо без наявності ***адміністратора системи*** (він же в різних системах іменується ***привілейованим користувачем*** або ***суперкористувачем***) або ж групи користувачів, що володіють правами адміністратора. Адміністратор призначає права іншим користувачам, а також має можливість у надзвичайних випадках одержати доступ до об'єктів будь-якого власника. Однак при цьому бажано, щоб дії адміністратора, як мінімум, фіксувалися системою з метою виявлення можливих зловживань з його боку.

3.5. Поділ файлів між процесами

У многозадачных ОС, а також у мережних системах, можлива ситуація, коли два або більш процеси намагаються одночасно використовувати той самий файл. Наприклад, два користувачі можуть одночасно працювати з одною базою даних. Будемо припускати, що з правами доступу усі в порядку, кожен процес окремо має право читати і записувати файл. Питання в тім, чи можна дозволити одночасну роботу, чи не приведе це до порушення цілісності даних.

Може привести. Якщо один процес обновляє дані у файлі, а іншої в цей час намагається

читати ці ж дані, то він може прочитати частково обновлені дані. Ще небезпечніше, коли два процеси намагаються одночасно змінити ті самі дані. У цьому випадку важко навіть пророчити, що в результаті буде збережено у файлі.

У принципі, завжди безпечними є лише два крайніх випадки:

123* тільки один процес працює з файлом, виконуючи читання і запис;

124* с файлом працює довільне число процесів, але усі вони виконують тільки читання.

ОС могла б забезпечити безпечний доступ, дозволяючи процесові відкривати файл тільки в цих двох випадках, тобто якщо файл не відкритий ще жодним іншим процесом або якщо файл відкритий кимсь тільки для читання і даний процес теж відкриває його для читання. Однак така суворість у ряді випадків істотно знизила б продуктивність системи. Скажемо, багато користувачів хотіли б одночасно працювати з одною базою даних. І в цьому немає нічого поганого, поки вони працюють з різними записами бази. Небезпека виникає тільки при одночасній роботі з однієї і тим же записом. Але ОС не може сама відстежити ситуацію так докладно, це може зробити програма, що керує базою даних. Через подібні ситуації, більшість ОС дозволяють програмам процесів самим визначати, чи допустимо спільний доступ у різних конкретних ситуаціях.

Типове рішення полягає в наступному. Прикладна програма, викликаючи системну функцію відкриття файлу, указує два додаткових параметри: режим доступу і режим поділу.

Режим доступу визначає, які операції сам процес збирається виконувати з файлом. Звичайно розрізняють доступ «тільки для читання», «тільки для запису», «для читання і запису».

Режим поділу визначає, які операції даний процес готовий дозволити іншим процесам, що захочуть відкрити той же файл. Зразковий набір режимів поділу – «заборона запису», «заборона читання», «заборона читання і записи» і «без заборон».

Перший процес, що відкриває файл, установлює за своїм розсудом режими доступу і поділу. Коли другий процес намагається відкрити той же файл, ОС перевіряє дві умови:

125* режим доступу другого процесу не повинний суперечити режимові поділу, установленому першим процесом;

126* режим поділу, запитуваний другим процесом, не повинний забороняти той режим доступу, що вже установив для себе перший процес.

У випадку порушення однієї з цих умов система не відкриває файл для другого процесу, функція відкриття файлу повертає помилку. Якщо ж умови дотримані, система відкриває файл для другого процесу, як би знімаючи із себе відповідальність за наслідки: ви цього хотіли – одержите.

Дотепер мова йшла тільки про поводження процесів і системи при відкритті файлу. Однак це не цілком вирішує проблему. Повернемося до приклада з базою даних. Нехай відповідний файл відкритий декількома процесами. Коли один із процесів приступає до роботи з конкретним записом, він повинний мати можливість тимчасово заборонити або обмежити доступ інших процесів до цього ж запису. Для цієї мети служить ***блокування*** процесом фрагментів файлу.

Для встановлення блокування процес викликає відповідну системну функцію, указуючи початок блокируемого фрагмента і його розмір. Якщо інший процес після цього спробує прочитати або записати дані, хоча б що частково попадають у заблокований фрагмент, то або функція читання (запису) видасть помилку, або процесові прийдеться чекати зняття блокування. Як правило, блокування встановлюється на як можна менший інтервал часу, щоб не знижувати продуктивність системи.

Описаний вище тип блокування називається ***ексклюзивним*** або винятковим блокуванням:

процес дозволяє собі і читання, і запис, а іншим процесам тимчасово забороняє те й інше. Деякі системи допускають також ***кооперативну*** (не виняткову) блокування: установлюючи неї, процес забороняє тільки запис усім процесам, у тому числі і собі самому, у той час як читання залишається дозволеним для усіх.

3.6. Файлова система FAT і керування даними в MS-DOS

3.6.1. Загальна характеристика системи FAT

Система FAT була розроблена для ОС MS-DOS. Це проста файлова система із сегментированным розміщенням, без многопользовательской захисту. Структура каталогів – деревоподібна, причому на кожнім дисковому томі створюється окреме дерево. Для вказівки місця розташування файлу може використовуватися його ***повне ім'я***, що містить букву диска, шлях по дереву каталогів і власне ім'я файлу, наприклад: «**`C:\UTILS\ARCH\RAR.EXE`**».

В ОС Windows також можливе використання FAT, особливо виправдане для дискет. Для твердих дисків великого обсягу система FAT стає малоефективної і поступово витісняється більш могутньою системою NTFS.

3.6.2. Структури даних на диску

При форматуванні дискети або роздягнула твердого диска в системі FAT весь дисковий простір розбивається на наступні області, показані на мал. 3- 2.

Структура диска в файловой системе FAT

Рис. 3- 2

127* ***BOOT-сектор*** містить основні кількісні параметри дискового тому і файлової системи, а також може містити програму початкового завантаження ОС.

128* ***таблиця FAT*** (File Allocation Table) – містить інформацію про розміщення файлів і вільного місця на диску. Через критичну важливість цієї таблиці вона завжди зберігається в двох екземплярах, що повинні бути ідентичні . Кожна операція, що змінює вміст FAT, повинна однаковим образом змінювати обидва екземпляри.

129* ***ROOT*** – кореневий каталог системи, що містить дані про файли і про підкаталоги верхнього рівня, кожний з яких у свою чергу може містити файли і підкаталоги.

130* ***Область даних*** – масив кластерів, що містить усі файли і всі каталоги (крім кореневого).

Розглянемо докладно, як зберігається вся інформація про файл, що мається в системі FAT.

При створенні файлу в одному з каталогів файлової системи створюється запис, що зберігає основний обсяг інформації про цей файл. Кожен каталог, крім кореневого, також є файлом особливого виду, і запис про нього утримується в батьківському каталозі. Каталожний запис завжди займає 32 байта, її структура показана в табл. 3.1.

Таблиця 3.1

Структура запису каталогу файлової системи FAT

Поле запису	**Розмір полючи (у байтах)**

Ім'я файлу	8
Розширення імені (тип файлу)	3
Атрибути (прапори)	1
Розмір файлу (у байтах)	4
Дата останньої зміни	2
Час останньої зміни	2
Резерв (не використовується)	10
Номер першого кластера файлу	2

Як видно з таблиці, ім'я файлу може займати не більш 8 символів плюс ще 3 символи розширення. На початку 80-х років здавалося, що цього цілком достатньо. Пізніше це обмеження охрестили «прокльоном 8 + 3», і избавить від нього файлову систему FAT удалося тільки в Windows 95.

Байт атрибутів містить набір битов, що характеризує властивості файлу. Поряд із практично марними атрибутами «схований», «системний» і «архівний», там утримуються і важливі: «тільки для читання», «каталог» і «мітка тому». Атрибут «тільки для читання» забороняє системі видаляти файл або відкривати його для запису. Атрибут «каталог» означає, що даний запис описує не звичайний файл, а каталог. Атрибут «мітка томи» може утримуватися тільки в кореневому каталозі, такий запис не описує ніякий файл, а замість цього містить у полях імені і розширення 11-символьну мітку (ім'я), привласнену даному дисковому той.

У цілому, запис каталогу містить майже усе, що системі відомо про файл, а якщо розмір файлу не перевищує одного кластера, те цілком усі. Якщо ж файл містить більш одного кластера, то номера інших можна знайти в таблиці FAT.

Таблиця FAT складається з записів, кількість яких дорівнює кількості кластерів в області даних, а розмір одного запису може бути дорівнює 12, 16 або 32 биткам. Відповідно говорять про різновиди файлової системи FAT-12, FAT-16 або FAT-32. Розмір запису повинний бути таким, щоб у ній можна було записати максимальний номер кластера. Наприклад, для стандартної тридюймової дискети ємністю 1.44 Мб досить використовувати FAT-12, оскільки це дозволяє мати $2^{12} = 4096$ кластерів (насправді, ледве менше), і навіть при кластерах розміром у 1 сектор (512 байт) цього більш ніж досить: $4096 \times 512 = 2$ Мб.

Запису FAT «по історичних причинах» нумеруються, починаючи з 2 і кінчаючи максимальним номером кластера, кожен запис FAT описує відповідний кластер з тим же номером. Запис може приймати наступні значення:

131* якщо кластер належить деякому файлові (або каталогові) і є останнім (або єдиним) у цьому файлі, то запис FAT містить спеціальне значення – всі одиниці (FFF_{16} для FAT-12 або $FFFF_{16}$ для FAT-16);

132* якщо кластер належить деякому файлові (або каталогові), але не є останнім у файлі, то запис FAT містить номер наступного кластера того ж файлу;

133* якщо кластер вільний, то запис містить усі нулі;

134* якщо кластер дефектний (тобто при перевірці диска з'ясувалося, що даний кластер містить хоча б один дефектний сектор), то запис містить спеціальне значення $FF7_{16}$ для FAT-12 або $FFF7_{16}$ для FAT-16.

Тепер ми знаємо, яким образом у системі FAT зберігається інформація про розміщення сегментированного файлу. Номер першого кластера файлу зберігається в записі каталогу, а інші кластери можна послідовно визначити по записах таблиці FAT.

У кожнім каталозі, крім кореневого, перші два записи містять спеціальні імена: ім'я «`.`» означає сам даний каталог, ім'я «`..`» – батьківський каталог.

3.6.3. Створення і видалення файлу

Щоб завершити вивчення структур дані системи FAT, розглянемо, як змінюються ці структури при операціях створення і видалення файлу.

Коли ОС виконує функцію створення файлу з заданим ім'ям у заданому каталозі, то вона, насамперед, знаходить даний каталог у дереві каталогів файлової системи. (Як вона його знаходить – опустимо для стислості. Спробуйте розібратися самі.) Прочитавши каталог, система перевіряє, немає чи в ньому вже запису з заданим ім'ям (тобто немає чи вже такого файлу). Якщо є, то чи не встановлений у цього файлу атрибут «тільки для читання»? Якщо встановлено, то новий файл створений бути не може, попередньо треба зняти атрибут. Якщо не встановлений, то старий файл віддаляється. Потім система знаходить у каталозі вільний запис. Якщо в каталозі немає вільного місця, то він може бути збільшений ще на один кластер, цей факт відбиває в таблиці FAT. Нарешті, знайшовши вільне місце, система заповнює поля запису про новий файл: його ім'я і розширення, дату і час останньої зміни, атрибути. Розмір і номер першого кластера встановлюються нульовими, тому що файл поки що не містить даних.

Надалі, при виконанні першої операції запису у файл, система знайде будь-як вільний кластер по таблиці FAT, відзначить його в FAT як останній кластер файлу, заповнить номер першого кластера в каталожному записі, виконає запис і занесе кількість записаних байт у поле розміру файлу в каталозі.

При видаленні файлу насамперед по каталожному записі перевіряється, чи можна його видалити (чине встановлений атрибут «тільки для читання»), а потім робляться дві речі:

135* перший байт імені файлу, що видаляється, заміняється на спеціальний символ з кодом E516 (він відображається як російська буква «»»; імовірно, розроблювачі системи FAT вважали, що цей код не може зустрітися в імені файлу);

136* усі записи таблиці FAT, що відповідають кластерам файлу, що видаляється, заповнюються нулями, тобто кластери з'являються вільними.

Як видно з описаної процедури, у момент видалення файлу його дані не стираються з диска, однак інформація про розміщення файлу псується. Якщо розмір файлу перевищував один кластер, то немає гарантії, що його удасться відновити у випадку ненавмисного видалення, навіть якщо користувач спохватиться відразу ж.

3.6.4. Робота з файлами в MS-DOS

3.6.4.1. Системні функції

Для роботи з файлами і каталогами системи FAT у MS-DOS передбачений досить багатий набір функцій. Усі вони викликаються за допомогою команди програмного переривання **`int 21h`**, а конкретна функція визначається числом, занесеним у регістр **`AH`**. Ці функції дозволяють, зокрема:

137* створювати файл, указуючи його повне ім'я;

138* видаляти файл;

139* змінювати атрибути файлу;

140* перейменовувати файл або переміщати його в інший каталог того ж диска ;

141* шукати в заданому каталозі усі файли, імена яких відповідають заданому шаблонові (наприклад, шаблонові «**`XYZ??.C*`**» відповідають усі файли, імена яких починаються з

«**XYZ**» і містять рівно 5 символів, а розширення починається з букви «**C**»);

142* створювати і видаляти каталоги;

143* задавати поточний диск і поточний каталог;

144* відкривати файл, одержуючи доступ до його даних, і закривати файл.

3.6.4.2. Доступ до даних

У MS-DOS існує два різних набори функцій, що дозволяють працювати з даними файлів. Один з них, заснований на використанні блоку керування файлом (FCB), навряд чи кимсь використовувався в останні 20 років, однак зберігається з розумінь сумісності з версією MS-DOS 1.0. Загальноприйнятий метод роботи з файлами заснований на використанні хэндлов .

Щоб відкрити існуючий файл, варто викликати відповідну функцію, указавши як параметри ім'я файлу і бажаний режим доступу (один із трьох: тільки читання, тільки запис, читання і запис). У більш пізніх версіях MS-DOS з'явилася також можливість указувати режим поділу, як розглянуто в п.3.5.Любою файл розглядається як послідовність байт, а якщо програма воліє розглядати файл як набір записів, то вона повинна сама вести перерахування номера запису в зсув (у байтах) від початку файлу. Операції читання і записи завжди виконуються від поточної позиції, що називається покажчиком читання/запису, і приводять до зсуву покажчика вперед на прочитану або записану кількість байт. Можливість довільного доступу до даних забезпечується операціями переміщення покажчика.

Хэндл – це деяке число, що система повертає користувальницькій програмі при удалому виконанні операції відкриття або створення файлу. Значення цього числа не грає ролі для програми. Важливо лише те, що при наступних звертаннях до відкритого файлу програма повинна передавати цей хэндл системі як покажчик на цей файл.

MS-DOS надає цілком достатній набір функцій для роботи з відкритими файлами. Сюди включаються функції читання і запису довільного числа байт, функція переміщення покажчика в довільну крапку файлу, функції установки і зняття блокування фрагментів файлу, примусового очищення кэш-буферов файлу (звичайно очищення виконується тільки при закритті файлу або при недостачі буферів, див. п.2.6.6;примусове очищення гарантує негайне збереження змін на диску).

При виклику функції відкриття файлу можна замість імені файлу вказати ім'я будь-якого символьного пристрою, наприклад, «**PRN:**». При цьому також повертається хэндл, з яким можна виконувати операції запису так само, як якби цей хэндл указував на дисковий файл. Зрозуміло, не можна відкрити принтер для читання, а також не можна виконувати переміщення покажчика назад.

При запуску будь-якої програми вона одержує «у подарунок» від MS-DOS п'ять уже відкритих хэндлов з номерами від 0 до 4. З них найбільш важливими є хэндл 0, що *по визначенню* вказує на стандартне введення програми, і хэндл 1 – стандартний висновок. Хэндл 2 означає стандартний пристрій для висновку повідомлень про помилки, хэндл 3 – стандартний пристрій послідовного введення/висновку (COM-порт), хэндл 4 – стандартний принтер.

Вище, у п. 2.2, давалося інше визначення стандартного введення і висновку, як пристроїв, використовуваних «за замовчуванням». Тут немає протиріччя. Компілятори мов програмування, зустрічаючи виклики процедур уведення/висновку без указівки файлу, транслюють них у виклики системних функцій MS-DOS з хэндлами, відповідно, 0 або 1.

Якщо програма запускається з командного рядка MS-DOS, то звичайно хэндл 0 указує на клавіатуру (точніше, на пристрій **CON:**), а хэндл 1 – на екран монітора (теж пристрій **CON:**, але працююче на висновок). Однак користувач може використовувати символи перенапрямку

стандартного введення (знак «**<**») і висновку (знаки «**>**» і «**>>**»). Наприклад, програма, запущена за допомогою команди «**MY_PROG <INFILE.TXT >PRN**» буде використовувати як стандартне уведення файл **INFILE.TXT**, а стандартний висновок направить на принтер. Знак «**>>**» означає додавання даних у кінець файлу стандартного висновку, знак «**>**» – перезапис файлу заново. Щоб виконати зазначене користувачем перенапрямок стандартного введення або висновку, система відкриває заданий файл або пристрій (а при знаку «**>>**» ще і виконує переміщення покажчика в кінець файлу) і забезпечує доступ до нього з програми, що запускається, через стандартний хэндл 0 або 1. Таким чином, що працює програма взагалі не знає, які саме пристрої або файли є її стандартним введенням і висновком.

Перенапрямок стандартного введення і висновку може бути виконано і програмою користувача. Звичайно це робиться перед тим, як дана програма запускає яку-небудь іншу програму, передаючи їй переспрямовані стандартні хэндлы.

3.6.4.3. Структури даних у пам'яті

Для забезпечення доступу до відкритих файлів MS-DOS використовує системні таблиці двох типів.

Таблиця SFT (System File Table) містить запису про усі файли, у даний момент відкритих програмами користувача і самої ОС. Ця таблиця зберігається в системній пам'яті, число записів у ній визначається параметром FILES у файлі конфігурації CONFIG.SYS, але не може перевищувати 255.

Якщо той самий файл був відкритий кілька разів (неважливо, однієї і тієї ж програм ой або різними програмами), то для нього буде кілька записів у SFT.

Кожен запис містить докладну інформацію про файл, достатню для виконання операцій з ним. Зокрема, у записі SFT утримуються:

145* копія каталожної інформації про файл;

146* адреса каталожного запису (сектор і номер запису в секторі);

147* поточне положення покажчика читання/запису;

148* номер останнього записаного або прочитаного кластера файлу;

149* адреса в пам'яті програми, що відкрила файл;

150* режим доступу, заданий при відкритті.

Крім того, у записі SFT утримується значення лічильника посилань на даний запис із усіх таблиць JFT, мова про які піде нижче. Коли цей лічильник стає рівним нулеві, запис SFT стає вільної, оскільки файл закритий.

На відміну від єдиної SFT, таблиці JFT (Job File Table) створюються для кожної програми, що запускається, тому одночасно може існувати кілька таких таблиць. (А відкіля в однозадачной MS-DOS можуть взятися одночасно кілька програм? Найпростіша відповідь: коли одна програма запускає іншу, то в пам'яті присутні обидві. Докладніше див. п. 4.4.3.) Таблиця JFT має найпростішу структуру: вона складається з однобайтовых записів, причому значення кожного запису являє собою індекс (номер запису) у таблиці SFT. Невикористовувані записи містять значення FF_{16}. Розмір таблиці за замовчуванням складає 20 записів (байт), але може бути збільшений до 255.

Тепер про хэндлах. Прикладна програма використовує хэндлы як деякі умовні номери відкритих файлів, конкретне значення хэндла при цьому не грає ніякої ролі (за зрозумілим виключенням стандартних хэндлов з 0 по 4). Насправді ж значення хэндла являє собою не що інше, як індекс запису в таблиці JFT даної програми. Перший запис таблиці відповідає хэндлу 0.

На мал. 3- 3 показаний зв'язок між хэндлами, таблицями JFT, таблицею SFT і відкритими

файлами/пристроями.

Таблицы JFT и SFT

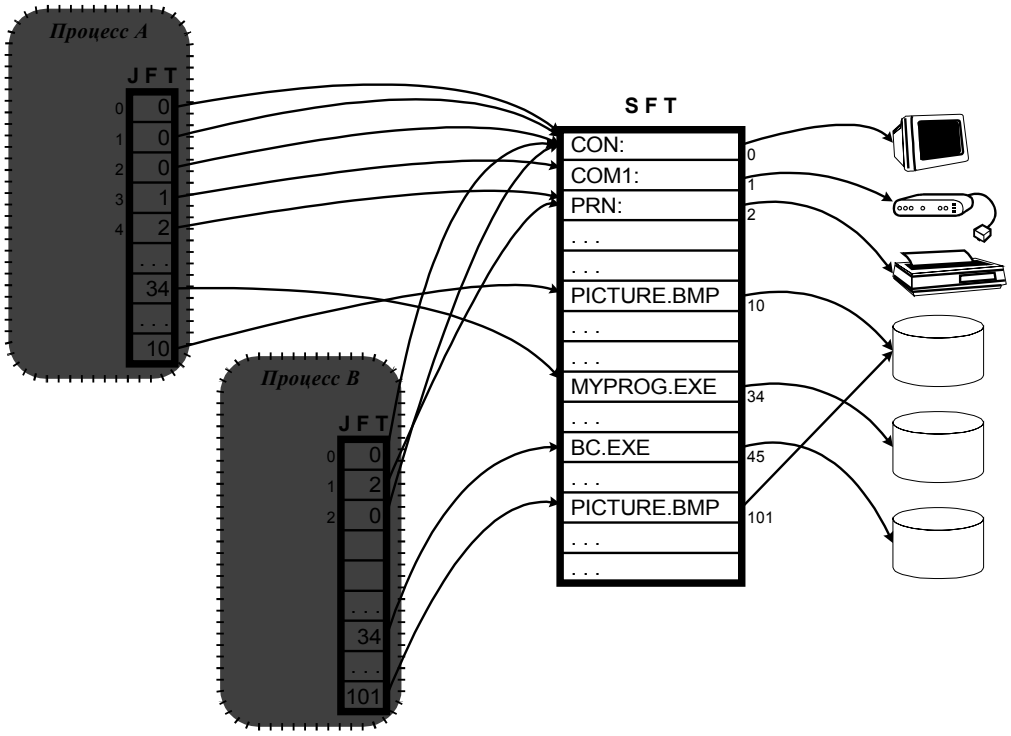


Рис. 3- 3

У прикладі, показаному на малюнку, стандартні хэндлы процесу А використовуються так, як це за замовчуванням робить MS-DOS: хэндлы 0, 1 і 2 указують на запис SFT, що відповідає консольному пристроєві CON, хэндл 3 – на запис про пристрій COM1, хэндл 4 –на запис про принтер. У процесу В стандартний висновок переспрямований на принтер, що відбито в значенні елемента 1 з JFT цього процесу. Хэндлы 3 і 4 для процесу В не показані, щоб не захламлять малюнок. Інші показані на малюнку елементи JFT обох процесів указують на записі SFT, що описують відкриті файли на дисках.

Помітимо, що з файлом PICTURE.BMP зв'язані два записи в таблиці SFT. Це означає, що даний файл був відкритий у кожнім процесі окремо (але, мабуть, з використанням одного з режимів поділу файлу).

Коли програма викликає яку-небудь із системних функцій і передає їй значення хэндла одного з відкритих файлів, то система знаходить адресу таблиці JFT програми, що викликала, читає зазначену хэндлом запис цієї таблиці, визначає відповідний індекс у таблиці SFT і одержує в такий спосіб доступ до інформації, необхідної для виконання операції з файлом.

У чому зміст такої двоступінчастої схеми? Чи не простіше було, щоб хэндл указував безпосередньо на запис SFT? Можна привести, принаймні, два очевидних аргументи на користь застосування JFT.

151* Що відбувається з файлами при завершенні програми, що них відкрила? Правила гарного програмістського тону вимагають, щоб програма перед закінченням роботи закрила за собою усі файли. Однак програміст може і не виконати цю вимогу, або програма може завершитися аварійно. У будь-якому випадку ОС повинна при завершенні програми закрити всі її файли. Як ОС довідається, які файли варто закрити? Відповідь дуже простій: досить переглянути

таблицю JFT програми, що завершується, і знайти там усі записи, відмінні від $_{FF}16$.

152* Використання JFT дає можливість відокремити логічне поняття стандартного пристрою (зокрема, стандартне введення – хэндл 0 і стандартний висновок – хэндл 1) від конкретних пристроїв. Перенапрямок стандартних пристроїв виконується шляхом зміни значень відповідних елементів JFT.

3.6.5. Нові версії системи FAT

Структури даних файлової системи є однієї з найбільш консервативних, що погано піддаються змінам характеристик ОС. Проблема полягає в тім, що при змінах структур даних важко зберегти сумісність з більш ранніми версіями. Було би катастрофою, якби вся безліч накопичених у світі файлів у системі FAT раптом перестало читатися в новій версії системи. Проте, деякі цікаві зміни все-таки удалася ввести при випуску версій Windows 95 і 98.

У Windows 95 було переборено прикре обмеження довжини імені файлу – знамените правило «8 + 3». Здавалося б, при розмірі запису каталогу в 32 байта важко сподіватися на довгі імена файлів. Проте, було знайдено забавне рішення цієї проблеми.

Розроблювачі з Microsoft звернули увагу, що ті записи каталогу, у яких зустрічається безглузда комбінація бітових атрибутів «схований + системний + тільки читання + мітка тому», просто-напросто ігноруються як системними програмами MS-DOS, так і розповсюдженими утилітами інших розроблювачів. Це дало можливість використовувати запису з такою комбінацією для збереження довгого імені файлу. Як і раніше для кожного файлу в каталозі мається основний запис у звичайному, старому форматі, що містить атрибути файлу, номер першого кластера й обов'язкове «коротке» ім'я. Однак якщо користувач при створенні файлу вказує ім'я, що не укладається в стандарт «8 + 3» або утримуючу малу літери, те перед основним записом буде вставлена потрібна кількість додаткових записів з розбитим на шматочки «довгим ім'ям» у кодуванні UNICODE (по 2 байти на символ, що дозволяє використовувати будь-як відомий алфавіт). Довжина імені, відповідно до документації, може досягати 255 символів (насправді, ледве менше).

Починаючи з Windows 98, з'явилася можливість використовувати новий різновид файлової системи – FAT-32. Її відмінність від FAT-12 і FAT-16 полягає не тільки в більшій розрядності номера кластера (хоча і це дуже важливо для великих дисків), але й у тім, що Microsoft той^-те-нарешті-те зважилася використовувати 10-байтовый резерв, що невідомо для яких цілей зберігався незайнятим у кожнім записі каталогу. Завдяки цьому з'явилася можливість додати до дати/часу останньої модифікації файлу ще два тимчасових штампи: дату/час створення (насправді, це дата/час останньої зміни каталожного запису) і дату останнього доступу до файлу.

3.7. Файлові системи і керування даними в UNIX

3.7.1. Архітектура файлової системи UNIX

Тут розглядається класична файлова система UNIX, називана іноді системою ***s5fs*** і підтримувана усіма версіями UNIX. Сучасні удосконалення файлової системи будуть розглянуті в п. 3.7.4.

3.7.1.1. Тверді і символічні зв'язки

Структуру каталогів файлової системи UNIX називають іноді мережний, щоб підкреслити її відмінність від строго ієрархічної (деревної) структури каталогів таких систем, як, наприклад, FAT. Відмінність це полягає в поняттях ***твердих і символічних зв'язків*** файлу.

Твердий зв'язок означає зв'язок між ім'ям файлу і самим файлом. Особливість UNIX у тім,

що будь-який файл може мати трохи (точніше, необмежена кількість) твердих зв'язків, тобто необмежена кількість імен. Це можуть бути різні імена в одному каталозі або навіть імена, що зберігаються в різних каталогах одного дискового тому.

Є чи яка-небудь користь від декількох імен одного файлу? Безумовно, є. Припустимо, користувач часто використовує яку-небудь системну програму або файл даних, лежачий десь глибоко в одній з галузей дерева каталогів. Замість того, щоб щораз указувати довгий шлях до потрібного файлу, користувач може просто створити новий твердий зв'язок, тобто дати файлові зручне ім'я і помістити це ім'я у свій особистий каталог. UNIX надає для цього команду **`link`**, що створює нове ім'я для зазначеного файлу.

Що відбудеться, якщо один з користувачів видалить ім'я файлу з каталогу? Відбудеться тільки обрив однієї з твердих зв'язків. Поки в даного файлу залишаються інші імена, файл продовжує існувати. Тільки після того, як вилучені усі імена файлу, система розуміє, що файл перестав бути доступний кому-небудь, і видаляє сам файл.

Усі тверді зв'язки (імена) одного файлу абсолютно рівноправні, серед них не можна виділити якесь «основне» ім'я.

Трохи іншим способом працює символічний зв'язок. Такий зв'язок являє собою файл, що містить тільки повне ім'я іншого файлу. Важливо при цьому те, що файл позначений у системі саме як символічний зв'язок, а не просто текстовий файл, що випадково зберігає ім'я файлу. Коли файл символічного зв'язку використовується як аргумент системної команди або функції, UNIX автоматично підставляє замість нього той файл, на який указує зв'язок.

Можна коротко сказати, що твердий зв'язок указує на сам файл, а символічна – на ім'я файлу. Обидва типи зв'язків проілюстровані на мал. 3- 4.

Жесткие и символические связи в файловой системе UNIX

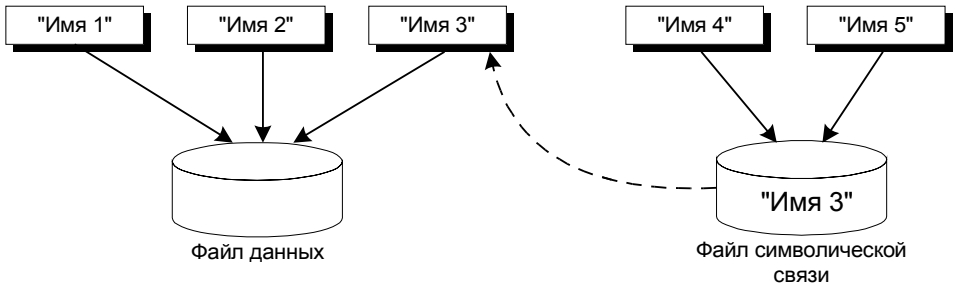


Рис. 3- 4

У прикладі на малюнку показаний файл даних, для якого маються три тверді зв'язки, тобто три імена в каталогах системи, позначені як «Ім'я 1», «Ім'я 2» і «Ім'я 3». Крім того, у системі мається файл типу «символічний зв'язок», що містить одне з імен файлу даних. Файл символічного зв'язку, як і будь-який інший файл, доступний по імені й у даному випадку має два імена (два твердих зв'язки): «Ім'я 4» і «Ім'я 5». Таким чином, використання кожного з п'яти імен у якості, наприклад, імені файлу, що відкривається, приведе до відкриття того самого файлу.

Припустимо, адміністратор системи вирішив замінити деякий файл його більш свіжою версією, залишивши те ж саме ім'я файлу. Якщо деякі користувачі зберігали тверді зв'язки на колишню версію, то вони так і будуть нею користуватися, поки явно не видалять її ім'я і не створять зв'язку на нову версію. Якщо ж користувач зберігав символічний зв'язок, то вона тепер буде вказувати на нову версію.

У Windows використовується деякий аналог поняття символічного зв'язку – ярлик файлу (shortcut).

Відмінність у тім, що з погляду файлової системи Windows ярлик не є якимсь особливим типом файлу, це звичайний текстовий файл із розширенням LNK. Ярлик розпізнається не файловою системою, а такими програмами, як Провідник (Explorer).

3.7.1.2. Монтируемые тому

У UNIX немає поняття «буква диска», подібно буквам **A:**, **C:** і т.д., використовуваним у MS-DOS і в Windows. У системі може бути кілька дискових томів, але, перш ніж одержати доступ до файлової системи будь-якого диска, крім основного, користувач повинний виконати операцію ***монтування диска***. Вона полягає в тім, що даний диск відображається на який-небудь з каталогів основного тому. Як правило, для цього використовуються порожні підкаталоги каталогу **/mount** або **/mnt**.

Якщо представити файлову систему на дисковому томі у виді дерева, то монтування тому – це як би «щеплення» одного дерева до якого-небудь місця на іншому, основному дереві. На відміну від цього, MS-DOS і Windows допускають використання декількох окремих дерев.

3.7.1.3. Типи й атрибути файлів

Для кожного файлу в UNIX зберігається його тип, що при видачі каталогу позначається одним з наступних символів:

- звичайний файл, тобто файл, що містить дані;

d каталог;

c символьний спеціальний файл, тобто, насправді, символьний пристрій;

b блоковий спеціальний файл;

l символічний зв'язок;

p іменований канал (буде розглянутий у п. 4.6.3);

s сокет – об'єкт, використовуваний для передачі даних по мережі.

Особливістю UNIX є те, що робота з різними типами об'єктів, перерахованими вище (файлами, пристроями, каналами, сокетами) організується з використанням того самого набору функцій файлового введення/висновку.

До числа атрибутів, що описують файл, відносяться його розмір у байтах, число твердих зв'язків і три «тимчасових штампи»: дата/час останнього доступу до файлу, останньому модифікації файлу, останньої модифікації атрибутів файлу. Цю останню величину часто називають неточно «датою створення файлу».

Для спеціальних файлів замість розміру зберігаються старший і молодший номери пристрою, див. п. 2.10.1.

Крім того, для кожного файлу зберігаються атрибути керування доступом, описані в наступному пункті, а також інформація про розміщення файлу на диску, описана в п. 3.7.2.

3.7.1.4. Керування доступом

Для кожного файлу (у тому числі каталогу, спеціального файлу) визначені такі поняття, як ***власник*** (один з користувачів системи) і ***група-власник***. Їхні числові ідентифікатори (називані,

відповідно, UID і GID) зберігаються разом з іншими атрибутами файлу. Повні імена, паролі й інші характеристики користувачів і груп зберігаються в окремому системному файлі. Власником файлу звичайно є той користувач, що створив цей файл.

Крім того, для файлу задаються ***атрибути захисту***, що зберігаються у виді 9 біт, що визначають допустимість трьох основних видів доступу – на читання, на запис і на виконання – для власника, для членів групи-власника і для інших користувачів.

При відображенні каталогу за допомогою команди **`ls -l`** ці атрибути показуються у виді 9 букв або прочерків, наприклад:

```
r w x  r - x  - - x
```

У приведеному прикладі показано, що сам власник файлу має усі права (r – читання, w – запис, x – виконання), члени групи-власника можуть читати файл і запускати на виконання (якщо цей файл містить програму), усім ладимо дозволене тільки виконання.

Привілейований користувач (адміністратор системи) завжди має повний доступ до усіх файлів.

У тому випадку, якщо файл є каталогом, права доступу на читання і на виконання розуміються трохи інакше. Право на читання каталогу дозволяє одержати імена файлів, що зберігаються в даному каталозі. Право на виконання каталогу означає можливість читати атрибути файлів каталогу, використовувати ці файли, а також право зробити даний каталог поточної. Можливі цікаві ситуації: якщо поточний користувач не має права на читання каталогу, але має право на його «виконання», те він не може довідатися імена файлів, що зберігаються в каталозі; однак, якщо він усе-таки якимсь образом довідався ім'я одного з файлів, то може відкрити цей файл або запустити на виконання (якщо цьому не перешкоджають атрибути доступу самого файлу).

Щоб видалити файл, не потрібно мати ніяких прав доступу до самого файлу, але необхідне право на запис у відповідний каталог (тому що видалення файлу є зміну не файлу, а каталогу). Утім, у сучасних версіях UNIX доданий ще один бітовий атрибут, при установці якого видалення дозволене тільки власникові файлу або користувачеві, що має право запису у файл.

Зміна атрибутів захисту, а також зміна власника файлу, можуть бути виконані тільки самим власником або привілейованим користувачем.

Ще два бітових атрибути, що мають відношення до захисту даних, називаються SUID і SGID. Вони визначають, які права (а точніше сказати, чиї ідентифікатори власника і групи) успадкує при запуску програма, що зберігається в даному файлі. Обидва біти за замовчуванням скинуті, при цьому програма використовує ідентифікатори UID і GID того користувача, що неї запустив. Програма як би «діє від імені цього користувача», і використовує його права доступу до файлів. Якщо ж для встановлені атрибути SUID і/або SGID, то запущена програма буде використовувати ідентифікатори UID і/або GID свого власника.

Для чого такі тонкості? Уявимо собі, що користувач хоче перемінити свій пароль. З одного боку – нормальне бажання, яке варто задовольнити. З іншого боку, пароль даного користувача зберігається в одному файлі з паролями інших користувачів, і доступ до цього файлу повинний бути, мабуть, закритий для всіх рядових користувачів. Можна підібрати й інші приклади ситуацій, коли звичайному користувачеві необхідний обмежений доступ до даних, що система повинна захищати.

Рішення, що пропонує UNIX, полягає в наступному. Для роботи з паролями мається спеціальна програма, власником якої є адміністратор системи. Програма доступна для виконання всім користувачам, але в неї встановлений біт SUID. У результаті цього запущена програма працює від імені адміністратора, тобто одержує необмежений доступ до файлів. Таким чином, відповідальність за захист системних даних перекладається на працюючу програму.

Слід зазначити, що в UNIX немає особливих засобів захисту для периферійних пристроїв. Як було описано вище, пристрою (а також іменовані канали і сокеты) вважаються особливими типами файлів, тому для них визначені ті ж атрибути захисту, що і для файлів.

3.7.2. Структури даних файлової системи UNIX

Дисковий тім UNIX складається з наступних основних областей, показаних (не в масштабі) на мал. 3- 5:

Структура диска в файловой системе UNIX (s5fs)

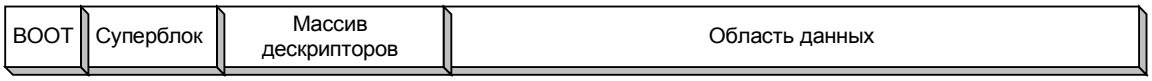


Рис. 3- 5

153* блок початкового завантаження (BOOT-сектор); його структура визначається не UNIX, а архітектурою використовуваного комп'ютера;

154* суперблок – містить основні зведення про дисковий том у цілому (розмір логічного блоку і кількість блоків, розміри основних областей, тип файлової системи, можливі режими доступу), а також дані про вільне місце на диску;

155* масив індексних дескрипторів, кожний з яких містить повні зведення про одному з файлів, що зберігаються на диску (крім імені цього файлу);

156* область даних, що складає з логічних блоків (кластерів), що використовуються для збереження файлів і каталогів (у UNIX використовується сегментированное розміщення файлів).

На відміну від системи FAT, де основні зведення про файл утримувалися в каталожному записі, UNIX використовує більш витончену схему.

Запис каталогу не містить *ніяких даних про файл, крім тільки* імені файлу і номера індексного дескриптора цього файлу.

У ранніх версіях UNIX кожен запис мав фіксовану довжину 16 байт, з яких 14 використовувалися для імені і 2 для номера. У більш сучасних версіях запис має перемінний розмір, що дозволяє використовувати довгі імена файлів.

Як і в системі FAT, у кожнім каталозі перші два записи містять спеціальні імена «`..`» (посилання на батьківський каталог) і «`.`» (посилання на даний каталог).

Точніше, це в FAT зроблене за прикладом UNIX.

Нульове значення номера відповідає вилученого запису каталогу.

Усі зведення про файл, крім імені, утримуються в його ***індексному дескрипторі*** (***inode***). Така схема робить можливими тверді зв'язки, описані вище: будь-яка кількість записів з одного каталогу або з різних каталогів може відноситися до тому самому файлу. Для цього треба тільки, щоб ці записи містили той самий номер inode.

Індексні дескриптори зберігаються в масиві, що займає окрему область диска. Розмір цього масиву задається при форматуванні, цей розмір визначає максимальна кількість файлів, яку можна розмістити на даному томі.

Дескриптор містить, насамперед, лічильник твердих зв'язків файлу, тобто число каталожних записів, що посилаються на даний дескриптор. Цей лічильник змінюється при створенні і видаленні зв'язків, його нульове значення говорить про те, що файл перестав бути доступним і повинний бути вилучений.

У дескрипторі утримуються тип і атрибути файлу, описані вище. Нарешті, тут же утримуються дані про розміщення файлу, що мають досить оригінальну структуру (див. мал. 3-6).

Информация о размещении файла

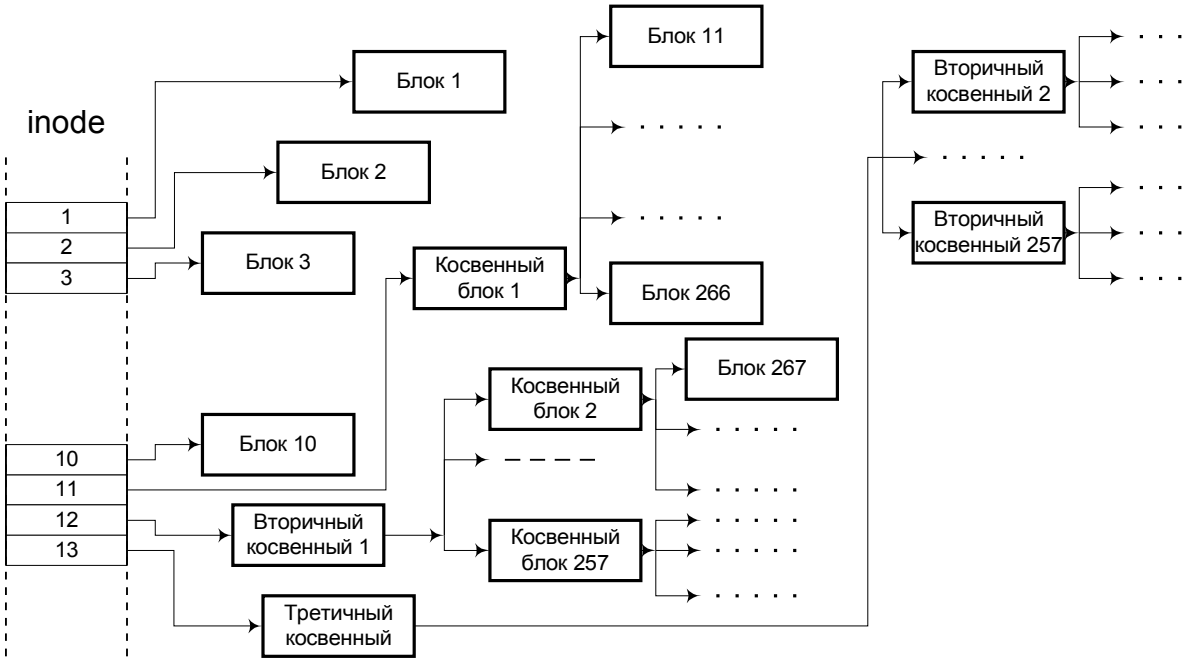


Рис. 3- 6

Розміщення блоків файлу задається масивом з 13 (у деяких версіях 14) елементів, кожний з яких може містити номер блоку в області даних. Нехай, для визначеності, блок дорівнює 1 Кб, а його номер займає 4 байти (обидві ці величини залежать від версії файлової системи). Перші 10 елементів масиву містять номера перших 10 блоків від початку файлу. Якщо розмір файлу перевищує 10 Кб, то в хід йде 11-й елемент масиву. Він містить номер ***непрямого блоку*** – такого блоку в області даних, що містить номера наступних 256 блоків файлу. Таким чином, використання непрямого блоку дозволяє працювати з файлами розміром до 266 Кб, використовуючи для этого один додатковий блок. Якщо файл перевищує 266 Кб, то в 12-ом елементі масиву утримується номер ***вторинного непрямого блоку***, що містить до 256 номерів непрямих блоків, кожний з яких…Ну, ви зрозуміли. Нарешті, для дуже великих файлів буде задіяний 13-й елемент масиву, що містить номер ***третинного непрямого блоку***, що вказує на 256 вторинних непрямих.

Підрахуйте, який максимальний розмір файлу може бути досягнуть при такій схемі адресації блоків.

Недоліком описаної схеми є те, що доступ до великих файлів вимагає значно більше часу, чим до маленького. Якщо розташування перших 10 Кб даних файлу записано безпосередньо в індексному дескрипторі, то для того, щоб прочитати дані, віддалені, скажемо, на 50 Мб від початку файлу, прийдеться спершу прочитати третинні, вторинний і звичайний непрямі блоки.

Ще одне важливе питання для будь-якої файлової системи – спосіб збереження даних про вільне місце. Для UNIX варто розрізняти два види вільних місць – вільні блоки в області даних і вільні індексні дескриптори, що бувають потрібні при створенні нових файлів. Кількість тих і інші може бути дуже великим. У суперблоці UNIX маються масиви для збереження деякої кількості номерів вільних блоків і вільних дескрипторів. Якщо вичерпані номери вільних

дескрипторів у суперблоці, то UNIX переглядає масив дескрипторів, знаходить у ньому вільні і виписує їхнього номера в суперблок.

Складніше обстоїть справа з вільними блоками даних. Перший елемент розміщеного в суперблоці масиву номерів вільних блоків указує на блок в області даних, що містить продовження цього масиву і, у першому елементі, покажчик на наступний блок продовження. Коли системі потрібні блоки дискової пам'яті, вона бере їхній з основного масиву в суперблоці, а при вичерпанні масиву читає його продовження в суперблок. При звільненні блоків відбувається зворотний процес: їхні номери записуються в масив, а при переповненні масиву весь його вміст листується в один з вільних блоків, номер якого заноситься в перший елемент масиву як адреса продовження списку. На мал. 3- 7 показана структура списку вільних блоків.

Структура списка свободных блоков

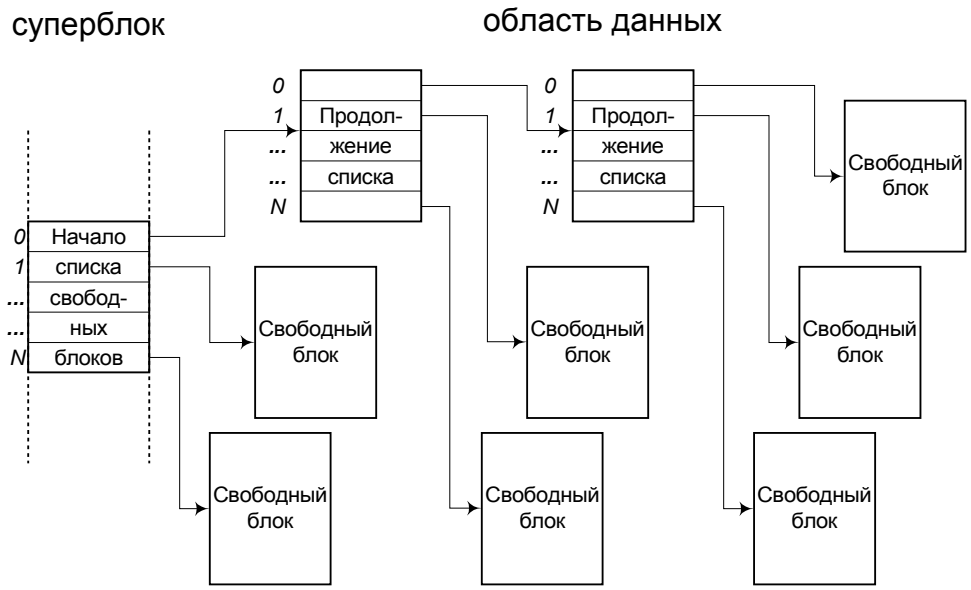


Рис. 3- 7

Як неважко зрозуміти, зі сказаного випливає, що блоки диска розподіляються «по стековому принципі»: блок, звільнений останнім, буде першим знову задіяний.

3.7.3. Доступ до даних у UNIX

UNIX надає в розпорядження прикладних програм набір системних викликів, що дозволяють виконувати основні операції з файлами і каталогами в цілому (створення, видалення, пошук, зміна власника і прав доступу), а також з даними, що зберігаються у файлах (відкриття і закриття файлу, читання і запис даних, переміщення покажчика у файлі).

В основі роботи з даними, як і для MS-DOS, лежить поняття хэндла відкритого файлу. Програма одержує значення хэндла при відкритті або створенні файлу, а потім використовує хэндл для посилання на відкритий файл при звертанні до функцій читання, запису, переміщення покажчика і т.п.

UNIX не має засобів керування поділом доступу при відкритті файлу, тобто завжди дозволяє декільком процесам відкривати той самий файл. Для забезпечення коректної роботи процеси можуть використовувати блокування фрагментів файлу. Можна установити блокування для запису (ексклюзивне блокування, див. п. 3.5) або для читання (кооперативне блокування). За замовчуванням у UNIX використовуються ***рекомендаційні*** блокування. Це означає, що система

не перешкоджає процесові звертатися до заблокованого фрагмента файлу. Процес повинний сам запитувати (якщо вважає потрібним), чи не заблокований даний фрагмент. У більш пізніх версіях UNIX стало можливим і ***обов'язковим*** блокуванням, при якому спроба звертання до заблокованого фрагмента приводить до помилки.

Для реалізації доступу до файлу за значенням хэндла в UNIX використовуються таблиці, аналогічні таблицям JFT і SFT у MS-DOS (див. п. 3.6.4.3). Однак, на відміну від MS-DOS, запис SFT не містить копії всіх атрибутів файлу. Замість цього UNIX зберігає в пам'яті окрему таблицю копій індексних дескрипторів (inode) усіх відкритих файлів. Запис SFT містить посилання на запис таблиці індексних дескрипторів, а поверх того – ті параметри, яких немає в inode: режим доступу до відкритого файлу, положення покажчика у файлі, кількість хэндлов, що вказують на даний запис SFT. Якщо той самий файл був відкритий кілька разів, то створюється кілька записів SFT, що вказують на той самий inode.

3.7.4. Розвиток файлових систем UNIX

Описана вище класична файлова система UNIX є по-своєму досить стрункою і могутньої, здатної задовольнити різноманітні вимоги користувачів. У той же час ця система давно вже зазнавала критики за два істотних недоліки, обумовлених вибором описаних вище структур даних. Ці недоліки наступні:

157* ненадійність;

158* низька продуктивність.

Поговоримо спочатку про ненадійність. Можна назвати наступні небезпечні місця в структурі файлової системи UNIX.

159* Широке використання зчеплених списковых структур для збереження життєво важливої інформації про розміщення файлів і вільних блоків. Програмістам відомий основний недолік зчеплених списків: при мінімальному перекручуванні хоча б одного з покажчиків губиться, у кращому випадку, уся наступна частина списку. Можлива і втрата даних, що не входять в ушкоджений список. Спробуйте уявити собі, що відбудеться на диску, якщо як номер непрямого блоку внаслідок апаратного збою буде записаний номер ні в чому не винного блоку даних із зовсім іншого файлу...

160* Інтенсивне використання суперблоку для збереження часто змінюється інформації. Оскільки кожна операція запису зв'язана з можливістю перекручування даних, під погрозою виявляються життєво важливі параметри файлової системи, що теж зберігаються в суперблоці, але не вимагають частих змін: розміри усієї файлової системи і її частин, розмір блоку і т.п.

161* Компактне розміщення на початку дискового тому найбільш важливих метаданных (суперблок, масив дескрипторів) приводить до того, що при механічному ушкодженні початкових доріжок диска губиться можливість доступу до всіх даних на диску.

Давно відомі і фактори, що впливають на зниження продуктивності роботи:

162* використання непрямих блоків приводить до того, що для пошуку потрібного місця у великому файлі можуть знадобитися додаткові операції читання з диска.

163* використовувані алгоритми виділення і звільнення дискових блоків за принципом «останнім звільнився – першим зайнятий» сприяють фрагментації дискового простору, а це, як нам відомо, приводить до зниження швидкості доступу.

Не дивно, що як заміну системи s5fs були запропоновані різні більш надійні і продуктивні файлові системи. До їхнього числа відносяться, наприклад, система FFS, використовувана у версії UNIX 4BSD, а також система ext2, що поставляється з Linux. Усі ці системи підтримують

звичну архітектуру UNIX, включаючи структуру каталогів, тверді і символічні зв'язки, типи файлів, набір атрибутів. Розходження полягають у реалізуючу цю архітектуру структурах дискових даних і в алгоритмах роботи з ними.

Постараємося дати загальне представлення про напрямки удосконалення файлових систем UNIX, не вдаючись у деталі конкретних систем.

164* Дисковий тім розбивається на кілька груп циліндрів. Кожна група містить копію загального суперблоку, а також окремий масив дескрипторів і блоки даних. У випадку ушкодження основного екземпляра суперблоку його вміст може бути відновлене по збереженій копії. Ушкодження метаданных в одній із груп циліндрів не веде до втрати інформації в інших групах.

165* Дані про кількість і місцезнаходження вільних блоків винесені із суперблоку і зберігаються окремо в кожній групі циліндрів.

166* Для збереження даних про вільні блоки використовується не списковая структура, а бітова карта, у якій кожен блок із групи циліндрів позначається як вільний або зайнятий. Це зменшує вага наслідків у випадку ушкодження даних.

167* Алгоритм розміщення файлів намагається *по можливості* розміщати усі файли одного каталогу, а також їхні дескриптори, у межах однієї групи циліндрів. Навпроти, підкаталоги по можливості не розміщаються в тій же групі циліндрів, що батьківський каталог. Таким способом звичайно вдається зменшити кількість переміщень голівок при роботі програми з файлами.

Потрібно відзначити, що в сучасних версіях UNIX, як і в інших сучасних ОС, підтримується одночасне використання різних файлових систем на різних дискових томах. Цієї мети служить поняття ***віртуальної файлової системи*** (VFS). Вона дозволяє підключати до єдиного дерева каталогів файлові системи різних типів, у тому числі такі чужорідні для UNIX, як FAT або NTFS. Для роботи з файлами використовується єдиний набір файлових функцій, однак при їхньому виклику, у залежності від того, до якої файлової системи належить даний файл, будуть викликатися різні підпрограми.

3.8. Файлова система NTFS і керування даними в Windows

3.8.1. Особливості файлової системи NTFS

Файлова система NTFS була розроблена спеціально для використання в ОС Windows NT як заміна для застарілої системи FAT. NTFS є основною системою і для нових версій – Windows 2000/XP.

Система NTFS спроектована як дуже могутня многопользовательская файлова система з великою кількістю можливостей. Проте, як затверджують розроблювачі, NTFS забезпечує більш швидкий доступ до даних, чим гранично проста система FAT, якщо обсяг диска перевищує 600 Мб.

Серед можливостей, відсутніх у FAT, але реалізованих у NTFS, можна назвати наступні.

168* Розвиті засоби захисту даних, що запобігають можливість несанкціонованого доступу до даних і при цьому позволяющие досить детально розмежувати права доступу для різних користувачів і груп користувачів.

169* Швидкий пошук файлів у великих каталогах.

170* Забезпечення цілісності даних у випадку збоїв або відключення харчування, засноване на механізмі транзакций. Це означає, що будь-яка операція з файлом розглядається як неподільна дія (транзакция), що повинна бути або виконане до кінця, або не виконано зовсім.

У ході операції система протоколює в спеціальному журналі хід виконання окремих етапів транзакции: запис даних, внесення змін у каталог і т.п. Якщо транзакция буде перервана на проміжному етапі, то при наступному завантаженні системи інформація з журналу дозволить «відкотити» недовиконану транзакцию, тобто скасувати виконані етапи.

171* Можливість стиску даних на рівні окремих файлів (тобто на одному дисковому томі можуть зберігатися файли як у стиснутому, так і в незжатому форматі).

172* Можливість збереження файлів у зашифрованому виді.

173* Механізм крапок повторного аналізу (reparse points), що дозволяє для окремих каталогів задати дії, що повинні виконуватися всякий раз, коли система звертається до даного каталогу. Зокрема, цей механізм дозволяє реалізувати такі UNIX-подібні можливості, як символічні зв'язки і монтування файлових систем.

174* Можливість протоколювання всіх змін, що відбуваються у файловій системі, таких як створення, зміна і видалення файлів і каталогів.

175* Розширюваність системи. Врахувавши важкий досвід, зв'язаний зі спробами модернізації FAT, розроблювачі NTFS заздалегідь заклали в систему можливість додавання нових, не передбачених у даний час атрибутів файлів.

Деякі можливості, закладені у файлову систему NTFS, навіть випереджають розвиток ОС Windows і поки не можуть бути використані в цій системі.

3.8.2. Структури дискових даних

Загальну структуру збереження даних на диску в системі NTFS часто характеризують двома короткими фразами:

На диску немає нічого, крім файлів.

У файлі немає нічого, крім атрибутів.

Поговоримо докладніше про ці загадкові твердження.

3.8.2.1. Головна таблиця файлів

Найбільш важливою частиною файлової системи на диску є ***головна таблиця файлів*** (MFT, Master File Table). Ця таблиця містить записи про усі файли і каталоги, розташованих на даному томі. Розмір запису складає один кластер, але не менш 1 Кб. Якщо метаданные про файл не містяться в одному записі, то можуть бути використані додаткові записи (не обов'язково сусідні).

Після форматування дискового тому, коли на ньому ще немає користувальницьких файлів, MFT містить 16 записів, з яких 11 містять опису файлів метаданных, а 5 зарезервовані як додаткові. Список файлів метаданных досить цікавий.

176* Перший запис MFT описує саму MFT, що теж вважається файлом. Це аж ніяк не формальність, оскільки MFT може, як та інші файли, складатися з декількох сегментів, розміщення яких задається в цьому записі.

177* Копія перших 16 записів MFT, що зберігається як файл де-небудь у середині диска. Це дозволяє відновити метаданные у випадку ушкодження основного екземпляра MFT.

178* Журнал протоколювання транзакций.

179* Файл інформації про том: ім'я тому, серійний номер, дата форматування і т.п.

180* Файл із перерахуванням всіх атрибутів, використовуваних для опису файлів на даному томі. Таким чином, список атрибутів не є жорстко фіксованим і може бути розширений у наступних версіях NTFS.

181* Корнєвій каталог тому.

182* Бітова карта зайнятості кластерів тому.

183* BOOT-сектор. Він як і раніше є першим сектором тому, але теж вважається файлом.

184* Файл, що складається з усіх дефектних кластерів на даному томі. Це дає підстави позначити в бітовій карті всі дефектні кластери як зайняті.

185* Файл, що містить усі різні дескриптори захисту, використовувані для файлів і каталогів даного тому (див. п. 3.8.4.2).

186* Файл, що задає пари прописних / малих літер для всіх мов, підтримуваних Windows. Такі дані необхідні, оскільки імена файлів можуть містити букви обох типів, але в Windows за традицією регістр букв в іменах файлів не розрізняється (у відмінність, наприклад, від UNIX).

187* Каталог, що містить ще 4 файли метаданных, доданих у Windows 2000. До них відносяться:

188* файл унікальних 16-байтовых ідентифікаторів, створюваних Windows для кожного файлу, на який мається ярлик або OLE-зв'язок; це дозволяє автоматично виправити ярлик, якщо вихідний файл був переміщений в інший каталог або навіть на інший комп'ютер у межах домена мережі;

189* файл квот дискового простору, виділюваних кожному користувачеві;

190* файл крапок повторного аналізу, установлених для каталогів даного тому;

191* файл журналу змін, що відбуваються на томі.

Далі, починаючи з 17-й позиції MFT, зберігаються записи метаданных про файли і каталоги, розміщених на даному томі.

Система намагається зберегти MFT безперервної, оскільки це прискорює звертання до всім описаним у ній файлам. Для цього система намагається по можливості не займати деяку область на початку диска під розміщення файлів, зберігаючи вільне місце для росту MFT.

3.8.2.2. Атрибути файлу

Кожен запис MFT містить набір атрибутів, що може розрізнятися для різних файлів і каталогів.

Атрибут у NTFS складається з заголовка і значення, а заголовок, у свою чергу, містить тип атрибута, його ім'я, довжину і дані про розміщення атрибута. Ім'я атрибута може отсутствовать, інші поля обов'язкові. Заголовок атрибута завжди зберігається в самому записі MFT, а значення – або теж у самому записі (при цьому атрибут називається ***резидентным***), або в кластері області даних (***нерезидентный*** атрибут). Деякі типи атрибутів зобов'язані бути резидентными, для інших типів вибір розміщення залежить від наявності достатнього вільного місця в записі MFT. Якщо атрибут нерезидентный, то в заголовку вказуються зведення про розміщення його значення на диску.

Розглянемо найбільш важливі типи атрибутів, використовуваних у записі про файл.

192* ***Ім'я файлу***. Цей атрибут завжди резидентен. Допускається кілька атрибутів цього типу, наприклад, «довге» ім'я (до 255 символів, включаючи букви будь-якої мови) і ім'я «8 + 3» для того ж файлу.

193* ***Стандартна інформація***. Це приблизно та інформація про файл, що зберігалася в записі каталогу FAT: розмір файлу, тимчасові штампи і бітові прапори.

194* ***Дескриптор захисту***. Він служить для завдання прав доступу до даного файлу для різних користувачів і груп, докладніше див. п. 3.8.4.2. У нових версіях NTFS запис MFT містить не

сам дескриптор, а посилання на його місце в системному файлі. Так виходить компактніше, оскільки звичайно на диску мається багато файлів з однаковими дескрипторами захисту і краще зберігати кожен дескриптор один раз, у спеціально відведеному для цього файлі метаданных.

195* ***Дані***. Це саме несподіване при першому знайомстві з NTFS: самі дані файлу розглядаються як один з типів атрибутів цього файлу. Наступна несподіванка полягає в тому, що атрибут даних невеликого файлу може зберігатися резидентно в складі запису MFT. Нагадаємо, що розмір цього запису – від 1 Кб і більше, так що місце для даних маленького файлу може найтися. Безумовно, резидентное збереження даних дозволяє прискорити доступ до них, оскільки запис MFT так чи інакше завжди читається при відкритті файлу.

Ще одна цікава особливість NTFS полягає в тім, що один файл може мати кілька атрибутів даних, визначальних кілька ***потоків даних*** (streams). Один з потоків безіменний, інші повинні мати імена. Виходить як би цілий каталог файлів усередині одного файлу. Безумовно, для цієї можливості можна придумати цікаві застосування, однак у жодній версії Windows, включаючи XP, поки не передбачені API-функції, що працюють з потоками даних.

Якщо запис MFT описує не файл, а каталог, то замість атрибута даних у ній утримується інший атрибут, що містить або весь каталог, або його частина. Якщо каталог занадто великий, то інші його частини зберігаються в нерезидентных атрибутах ще одного типу. Тут ми не будемо розглядати це питання детально, однак слід зазначити, що атрибути, що описують великий каталог, утворять структуру даних, відому як Б-дерево (B-tree). Ця структура дозволяє прискорити пошук файлу в каталозі.

Запис каталогу містить лише ім'я файлу, номер запису про цей файл у MFT і копію атрибута «стандартна інформація». Ця копія дозволяє відображати вміст каталогу без читання записів MFT про кожен файл.

При порівнянні структури NTFS з раніше розглянутою структурою s5fs можна знайти деяку аналогію між таблицею MFT і масивом індексних дескрипторів, що містять всю інформацію про файл у s5fs. При цьому NTFS має значно більш складну структуру і надає багато додаткових можливостей.

3.8.3. Доступ до даних

Windows надає прикладним програмам API-функцію **`CreateFile`**, що може використовуватися як для створення нового файлу, так і для відкриття існуючого. У будь-якому випадку ця функція створює в системній пам'яті об'єкт типу «відкритий файл», саме тому назва функції починається зі слова «Create».

Функція **`CreateFile`** може використовуватися для роботи з файлами будь-якої файлової системи, підтримуваної Windows (зокрема, FAT і NTFS).

Параметри цієї функції численні і дають досить гарне представлення про можливості роботи з файлами в Windows. Деякі параметри мають сенс тільки для NTFS (але не для FAT) або тільки для Windows NT (але не для Windows 95). Список параметрів містить у собі наступні параметри.

196* Ім'я файлу (включаючи шлях до каталогу, якщо файл розташований не в поточному каталозі). Замість імені файлу може також бути зазначене спеціальне ім'я пристрою, у тому числі навіть ім'я фізичного або логічного диска.

197* Режим доступу. Може бути зазначений доступ для читання, для запису або їхня комбінація.

198* Режим поділу. Він може містити в собі дозвіл іншим процесам читати файл, записувати

дані у файл, видаляти файл або будь-яку комбінацію цих дозволів, у тому числі, зрозуміло, і відсутність усіх дозволів.

199* Атрибути захисту. Їхнє використання буде описане в п. 3.8.4.2.

200* Режим створення. Визначає дії функції у випадках, коли файл із заданим ім'ям вже існує і коли не існує. Визначено наступні режими.

201* **CREATE_NEW** – Створюється новий файл. Якщо файл вже існує, видається помилка.

202* **CREATE_ALWAYS** – Створюється новий файл у будь-якому випадку, навіть якщо файл із таким ім'ям вже існує.

203* **OPEN_EXISTING** – Відкривається існуючий файл. Видає помилку, якщо файл не існує.

204* **OPEN_ALWAYS** – Якщо файл існує, те він відкривається, якщо не існує – створюється новий файл.

205* **TRUNCATE_EXISTING** – Відкривається існуючий файл, але весь його уміст віддаляється. Якщо файл не існує, видається помилка.

206* Великий набір атрибутів і прапорів, якому варто розглянути докладніше. У даному випадку атрибутами називаються ознаки файлу, що встановлюються при його створенні, а прапорами – ознаки, що уточнюють режим роботи з відкритим файлом.

До атрибутів файлу відносяться всі ті, котрі Windows успадкувала від MS-DOS (тільки для читання, схований, системний, архівний), а також атрибут «стиснутий» (тобто файл, створюваний у NTFS, буде зберігатися в стиснутому виді) і атрибут «тимчасовий». Цей атрибут означає, що файл, імовірно, буде незабаром вилучений, а тому система повинна спробувати удержати його дані в пам'яті, не витрачаючи даремно час на запис файлу на диск.

Прапори функції надають, зокрема, що випливають можливості:

207* при операціях запису негайно виконувати запис на диск (очищати кэш-буфера файлу);

208* узагалі виключити використання кэша для даного файлу, завжди записувати і читати сектори даних безпосередньо з диска;

209* указати системі бажаність оптимальної буферизации для послідовного доступу або, навпаки, для довільного доступу;

210* відкрити файл для виконання асинхронних операцій;

211* указати системі, що файл повинний бути автоматично вилучений відразу ж, як тільки він буде закритий.

Функція **CreateFile** повертає хэндл відкритого файлу. Цей хэндл може потім використовуватися при звертанні до функцій читання, запису, переміщення покажчика, очищення буферів, блокування фрагментів, закриття фала й ін.

Читання і запис даних при синхронних операціях починається з поточної позиції покажчика і супроводжується зсувом покажчика читання/запису вперед на кількість прочитаних/записаних байт. Однак якщо при відкритті файлу був зазначений прапор асинхронних операцій, то покажчик не використовується. Замість цього при кожнім виклику функції читання або записи повинний задаватися додатковий параметр – зсув від початку файлу тих даних, який варто прочитати або записати.

Як ви думаєте, чому при асинхронних операціях не використовується покажчик читання/запису?

Процес, що запустив асинхронну операцію читання/запису, може потім перевірити її результат за допомогою виклику системної функції, що, у залежності від параметрів, або очікує завершення операції, або просто перевіряє, чи завершилася вона. Крім того, їсти можливість зв'язати з завершенням асинхронної операції або подія (event), або функцію завершення, що буде

викликана, якщо операція довершена, а нитка процесу, що почала операцію, викликала функцію чекання (див. пп.4.5.5.2,4.5.5.3про події і функції чекання).

Деякі файлові функції не вимагають хэндла, тобто виконуються над закритими файлами. Сюди відносяться видалення файлу, копіювання, перейменування, переміщення файлу, пошук файлу по заданому шляху і шаблонові імені і т.п. Як цікаві особливості файлових операцій Windows можна відзначити наступні.

212* Операції копіювання файлу або переміщення його на інший диск можна виконати викликом однієї системної функції, навіть не відкриваючи файл. В інших ОС подібні операції вимагають написання цілої процедури.

213* Спеціально для заміни версій системних програм передбачений варіант перейменування/переміщення файлу з відкладанням фактичного виконання до перезавантаження системи. Іншим способом неможливо було б, наприклад, установити нові версії системних бібліотек, оскільки існуючі версії постійно відкриті і тому не можуть бути вилучені інакше як при старті системи.

3.8.4. Захист даних

Засобу безпеки в Windows NT/2000/XP являють собою окрему підсистему, що забезпечує захист не тільки файлів, але й інших типів системних об'єктів. Файли і каталоги NTFS являють собою найбільш типові приклади об'єктів, що захищаються.

Як відомо, Windows дозволяє використовувати різні файлові системи, при цьому можливості захисту даних визначаються архітектурою конкретної файлової системи. Наприклад, якщо на дисковому томі використовується система FAT (де, як нам відомо, ніяких засобів захисту не передбачене), те Windows може хіба що обмежити доступ до усього того, але не до окремих файлів і каталогів.

3.8.4.1. Аутентификация користувача

Важливим елементом будь-якої системи захисту даних є процедура входу в систему, при якій виконується аутентификация користувача. У Windows NT для виклику діалогу входу в систему використовується відома «комбінація з трьох пальців» – **`Ctrl+Alt+Del`**. Як затверджують розроблювачі, ніяка «троянська» програма не може перехопити обробку цієї комбінації і використовувати неї з метою колекціонування паролів.

Чи не хоче хто-небудь спробувати?

Система шукає введене ім'я користувача спочатку в списку користувачів даного комп'ютера, а потім і на інших комп'ютерах поточного домена локальної мережі. У випадку, якщо ім'я знайдене і пароль збігся, система одержує доступ до ***облікового запису*** (account) даного користувача.

На підставі зведень з облікового запису користувача система формує структуру даних, що називається ***маркером доступу*** (access token). Маркер містить ідентифікатор користувача (SID, Security IDentifier), ідентифікатори всіх груп, у які включений даний користувач, а також набір ***привілеїв***, якими володіє користувач.

Привілеями називаються права загального характеру, не зв'язані з конкретними об'єктами. До числа привілеїв, доступних тільки адміністраторові, відносяться, наприклад, права на установку системного часу, на створення нових користувачів, на присвоєння чужих файлів. Деякі скромні привілеї звичайно надаються всім користувачам (наприклад, такі, як право налагоджувати процеси, право одержувати повідомлення про зміни у файловій системі).

У подальшій роботі, коли користувачеві повинний бути наданий доступ до яких-небудь ресурсів, що захищаються, рішення про доступ приймається на підставі інформації з маркера

доступу.

3.8.4.2. Дескриптор захисту

Для будь-якого об'єкта, що захищається, Windows (файлу, каталогу, диска, пристрою, семафора, процесу і т.п.) може бути задана спеціальна структура даних – атрибути захисту.

Основним змістом атрибутів захисту є інша структура – ***дескриптор захисту***. Цей дескриптор містить наступні дані:

214* ідентифікатор захисту (SID) власника об'єкта;

215* ідентифікатор захисту первинної групи власника об'єкта;

216* користувальницький («дискреційний», «розмежувальний») список керування доступом (DACL, Discretionary Access Control List);

217* системний список керування доступом (SACL, System Access Control List).

Користувальницький список керує ***дозволами*** і ***забороонами*** доступу до даного об'єкта. Змінювати цей список може тільки власник об'єкта.

Системний список керує тільки ***аудитом*** доступу даному об'єктові, тобто задає, які дії користувачів стосовно даного об'єкта повинні бути запротокольовані в системному журналі. Змінювати цей список може тільки користувач, що має права адміністратора системи.

У Windows, на відміну від багатьох інших ОС, адміністратор не всесильний. Він не може заборонити або дозволити кому б те ні було, навіть самому собі, доступ до чужого файлу. Інша справа, що адміністратор має право оголосити себе власником будь-якого файлу, але потім він не зможе повернути файл колишньому хазяїнові. Подібні обмеження випливають з розуміння, що адміністратор теж не завжди ангел і, хоча він повинний мати в системі великі права, його дії варто хоч якось контролювати.

Обидва списки керування доступом мають однакову структуру, їх основною частиною є масив ***записів керування доступом*** (ACE, Access Control Entity).

Розглянемо структуру запису ACE. Вона містить:

218* тип ACE, що може бути одним з наступних: дозвіл, заборона, аудит;

219* прапори, що уточнюють особливості дії даної ACE;

220* бітова маска видів доступу, що вказує, які саме дії варто дозволити, заборонити або піддати аудитові;

221* ідентифікатор (SID) користувача або групи, чиї права визначає дана ACE.

Більш цікавий користувальницький список. Він може містити тільки запису дозволу і заборони. На початку списку завжди йдуть записи, що забороняють, що потім дозволяють.

Коли користувач запитує доступ до об'єкта (тобто, наприклад, програма, запущена цим користувачем, викликає функцію відкриття файлу), відбувається перевірка прав доступу. Вона виконується на основі порівняння маркера доступу користувача зі списком DACL. Система переглядає один по одному всі записи ACE з DACL, для кожної ACE визначає записаний у ній SID і звіряє, чи не є він ідентифікатором поточного користувача або однієї з груп, куди входить цей користувач. Якщо ні, то дана ACE не має до нього відносини і не враховується. Якщо так, то виконується порівняння праве, необхідних користувачеві для виконання запитаної операції з маскою видів доступу з ACE. При цьому права аналізуються досить детально: наприклад, відкриття файлу на читання має на увазі наявність прав на читання даних, на читання атрибутів (у тому числі власника й атрибутів захисту), на використання файлу як об'єкта синхронізації (див. п. 4.5.5.2).

Якщо в що *забороняє* ACE найдеться хоча б один одиничний біт у позиції, що відповідає одному з запитаних видів доступу, то вся операція, почата користувачем, вважається забороненої і подальші перевірки не виробляються.

Якщо такі біти будуть знайдені в що *дозволяє* ACE, то перевірка наступних ACE виконується доти, поки не будуть дозволені і всі інші запитані види доступу.

Як видумаєте, чому в списку DACL спочатку йдуть що забороняють ACE, а тільки що потім дозволяють?

Таким чином, говорячи коротко, користувач одержить доступ до об'єкта тільки в тому випадку, якщо всі запитані їм види доступу явно дозволені і жоден їхній них не заборонений.

В роки перебудовии багато писалося про дв протилежні принципи: «заборонене ус, що не дозволено» або «дозволено ус, що не заборонено». У Windows усі набагато суворіше: заборонено усе, що заборонено, і усе, що не дозволено.

Список DACL із усіма необхідними дозволами і заборонами може бути встановлений програмно при створенні файлу, а згодом програмно ж може бути змінений власником. Можна також змінювати дозволу в діалозі, скориставшись вікном властивостей файлу.

Маються також два крайніх випадки. Список DACL може зовсім отсутствовать (для цього досить, наприклад, при створенні файлу вказати NULL замість атрибутів захисту), при цьому права доступу не перевіряються, усі дії дозволені всім користувачам. Список DACL може бути присутнім, але мати нульову довжину (немає ні однієї ACE). Як випливає з загальних правил, у цьому випадку в доступі буде відмовлене усім, у тому числі і хазяїнові файлу.

4. КЕРУВАННЯ ПРОЦЕСАМИ

4.1. Основні задачі керування процесами

Під керуванням процесами розуміються процедури ОС, що забезпечують запуск системних і прикладних програм, їхнє виконання і завершення.

В однозадачных ОС керування процесами вирішує наступні задачі:

222* завантаження програми в пам'ять, підготовка її до запуску і запуск на виконання;

223* виконання системних викликів процесу;

224* обробка помилок, що виникли в ході виконання;

225* нормальне завершення процесу;

226* припинення процесу у випадку помилки або втручання користувача.

Усі ці задачі вирішуються порівняно просто.

У многозадачном режимі додаються значно більш серйозні задачі:

227* ефективна реалізація рівнобіжного виконання процесів на єдиному процесорі, переключення процесора між процесами;

228* вибір чергового процесу для виконання з урахуванням заданих пріоритетів процесів і статистики використання процесора;

229* виключення можливості несанкціонованого втручання одного процесу у виконання іншого;

230* запобігання або усунення тупикових ситуацій, що виникають при конкуренції процесів за системні ресурси;

231* забезпечення синхронізації процесів і обміну даними між ними.

4.2. Реалізація многозадачного режиму

4.2.1. Поняття процесу і ресурсу

Відповідно до визначення, даному в /7/, «*послідовний процес (іноді називаний «задача») є робота, вироблена послідовним процесором при виконанні програми з її даними*».

Проаналізуємо це визначення. Воно підкреслює послідовний характер процесу, тобто виконання команд у визначеному порядку. Термін «задача» ми будемо розуміти як синонім терміна «процес» (у деяких ОС ці терміни розрізняються). Далі, процес – поняття динамічне. Програма – це текст, процес – виконання цього тексту. Звичайно, на практиці ми часто говоримо: «програма викликає функцію», «програма чекає введення» і т.п., однак, строго говорячи, вірніше був б «процес, що виконує програму, викликає...».

Ще один важливий момент у визначенні – згадування даних. У многозадачных системах найчастіше та сама програма може запускатися кілька разів (наприклад, можна кілька разів відкрити текстовий редактор Notepad для різних файлів). Це означає, що кілька процесів можуть використовувати ту саму програму, але з різними даними.

При описі роботи многозадачных систем основна увага приділяється питанням, зв'язаним з рівнобіжним виконанням, тобто з одночасною роботою декількох процесів. Однак загальний термін «рівнобіжне виконання» поєднує два істотно різних способи організації виконання процесів – ***синхронний*** і ***асинхронний*** паралелізм.

Синхронний паралелізм припускає наявність загальної тактової послідовності, що керує кроками виконання паралельно працюючих процесів. Синхронна організація використовується в деяких типах многопроцессорных обчислювальних систем.

При асинхронному паралелізмі ніякого загального такту немає. Процеси виконуються незалежно друг від друга, при цьому не робиться ніяких припущень про їхню порівняльну швидкість, про співвідношення часу виконання різних фрагментів програм і т.п. Зіставити просування різних процесів можна тільки в явно заданих крапках програми процесів, називаних ***крапками синхронізації***. Синхронізація звичайно означає чекання одним процесом якої-небудь події, зв'язаного з іншим процесом. Наприклад, після розгалуження виконання на дві рівнобіжні галузі і виконання галузями визначених задач може знадобитися звести галузі воєдино: та галузь, що закінчила своє виконання раніш, повинна дочекатися завершення іншої галузі. Інший приклад: процес, що виконує обробку введених рядків, може чекати, поки інший процес не прочитає черговий рядок з файлу або з клавіатури.

Крапка синхронізації може бути зв'язана також з обміном даними між процесами. Коли приймач^-приймач-процес-приймач завершує прийом даних, то можна бути, принаймні, упевненим, що джерело^-джерело-процес-джерело досягло того місця в програмі, де він повинний був передати дані.

Надалі усюди буде розглядатися тільки асинхронний паралелізм, оскільки він характерний для роботи ОС.

Іншим основним поняттям, тісно зв'язаним з керуванням процесами, є поняття ***ресурсу***. Під ресурсом розуміється будь-який апаратний або програмний об'єкт, що може знадобитися для роботи процесів і доступ до якого може при цьому викликати конкуренцію процесів. Говорячи спрощено, ресурс – це щось дефіцитне в обчислювальній системі. До найважливіших ресурсів будь-якої системи відносяться процесор (точніше сказати, процесорний час), основна пам'ять, периферійні пристрої, файли. У залежності від конкретної ОС, до дефіцитних ресурсів можуть відноситися місця в таблиці процесів або в таблиці відкритих файлів, буферы кэша, блоки у файлі підкачування й інші системні структури даних.

Якщо два процеси ніяк не зв'язані логікою своєї роботи, то ніякої синхронізації або обміну даними між ними ні, однак такі процеси все-таки можуть впливати один на одного внаслідок ***конкуренції за ресурси***. Многозадачная ОС керує доступом процесів до ресурсів. У деяких випадках система тимчасово закріплює ресурс за одним процесом, відмовляючи іншим процесам у доступі або змушуючи них чекати звільнення ресурсу. В інших випадках виявляється можливим спільний доступ декількох процесів до одного ресурсу.

4.2.2. Квазипараллельное виконання процесів

З погляду зовнішнього спостерігача, у гарної многозадачной ОС відбувається одночасна, рівнобіжна робота декількох процесів. Однак зрозуміло, що ця одночасність удавана. Насправді, якщо в системі працює лише один процесор, то в кожен момент часу він виконує команди, що відносяться тільки до одному з наявних процесів. Ілюзія паралельності створюється за рахунок того, що процеси переміняють один одного через малі інтервали часу, що людина-спостерігач не в силах відстежити. Подібна організація роботи називається ***квазипараллельным*** виконанням процесів.

Зрозуміло, якщо в системі мається кілька процесорів, то може бути організоване дійсне рівнобіжне виконання процесів, кількість яких не перевищує кількості процесорів. При більшому числі процесів може використовуватися змішана організація, що сполучить щиру паралельність і квазипараллельность.

Важливо відзначити, що для більшості задач взаємодії процесів немає різниці, якого роду паралельність використовується в даної ОС. Узагалі, основні проблеми керування процесами можна розбити на два рівні:

232* проблеми коректної й ефективної реалізації рівнобіжного (тобто звичайно квазипараллельного) виконання процесів – це проблеми нижнього рівня;

233* проблеми коректної взаємодії рівнобіжних процесів – це проблеми верхнього рівня, при розгляді яких вважається, що низкоуровневые проблеми реалізації процесів так чи інакше вирішені.

Така розбивка полегшує проектування і налагодження систем, а також дозволяє краще зрозуміти істота розглянутих проблем.

4.2.3. Стану процесу

Любою процес у многозадачной ОС багаторазово випробує перехід з одного ***стану*** в інше.

Основних станів всего три.

234* ***Робота*** (running) – у цьому стані знаходиться процес, програму якого в даний момент виконує процесор. Працюючий процес іноді зручно називати також ***поточним*** процесом.

235* ***Готовність*** (ready) – стан, їх якого процес може бути переведений у стан роботи, як тільки це рахує потрібним зробити ОС.

236* ***Блокування*** або, що той же саме, ***сон*** (sleeping, waiting) – стан, у якому процес не може продовжувати виконання, поки не відбудеться деяке *зовнішнє* стосовно процесу подія.

Перші два стани часто поєднують поняттям ***активного*** стану процесу.

Для станів готовності і сну загальне те, що процес не працює. У чому різниця між цими двома «способами не працювати»?

Готовий до виконання процес не виконується тільки тому, що є інші не менш готові процеси, на думку системи більш гідні займати зараз процесорний час. У кожен момент часу вибір одного з готових процесів на роль працюючих визначається логікою роботи ОС. Цей вибір повинний забезпечувати ефективну квазипараллельную роботу готових процесів. Як

вирішується ця задача – буде розглянуто нижче.

На відміну від цього, що спить процес – це завжди процес, що очікує деякої конкретної події. Сплячий процес не зможе заробити, навіть якщо процесор раптом виявиться вільним. Такий процес, у відповідності зі своєю власною логікою, чекає чогось, що повиннео відбутися.

Чого він може чекати? Ну, наприклад:

237* завершення початої операції синхронного введення/висновку (тобто, наприклад, процес чекає натискання клавіші **`Enter`** або закінчення запису на диск);

238* звільнення запитаного в системи ресурсу (наприклад, додаткової області пам'яті або відкритого файлу);

239* витікання заданого інтервалу часу («*посплю-ка я хвилин десять!*») або досягнення заданого моменту часу («*розбудите мене рівно опівночі!*») (в обох випадках процес чекає сигналу від запрограмованого таймера);

240* сигналу на продовження дій від іншого, взаємозалежного процесу;

241* повідомлення від системи про необхідність виконати визначені дії (наприклад, перемалювати уміст вікна).

У кожнім з названих (і багатьох неназваних) випадків повинне відбутися деяка подія, джерело якого лежить *поза* даним процесом.

Чисто умовно можна сказати, що якби в обчислювальну систему раптом було додано ще кілька процесорів, те «готові» процеси могли б відразу перейти в стан «робота», але «сплячі» продовжили б свій сон.

Зрозуміло, як ми бачили в п. 2.5, процес може виконувати чекання шляхом циклічної перевірки очікуваної умови. При цьому він формально буде залишатися активним, розтрачуючи дорогоцінний процесорний час на те, що в п. 2.5.2 було названо ***активним чеканням***. Однак таке рішення буде говорити лише про волаючу некваліфікованість програміста. Будь-яка многозадачная ОС надає в розпорядження прикладних програм набір функцій, що переводять їхній процес, що викликав, у стан сну, у якому процес не намагається використовувати процесорний час (іншими словами, стан сну є ***стан пасивного*** чекання). Такі системні функції називаються що ***блокують***. До їхнього числа відносяться функції синхронного введення/висновку, запиту ресурсів, припинення до заданого часу, одержання повідомлень і багато хто інші.

Оскільки ОС бере на себе блокування, «приспання» процесу, воно повиннао забезпечити і його розблокування, «пробудження». Щоб це стало можливим, система повинна для кожного сплячого процесу пам'ятати, «чого він чекає», тобто пам'ятати умови пробудження процесу. Система відслідковує всі події, здатні розблокувати який-небудь процес (у багатьох випадках використовуючи для цього апаратні переривання) і, коли для одного або відразу декількох процесів настає очікувана подія, переводить ці події зі стану сну в стан готовності.

На мал. 4- 1 показані основні стани процесу і переходи між ними. Цей малюнок кочує з книги в книгу, оскільки він дійсно наочно відбиває саму суть роботи многозадачных систем.

Основные состояния процесса

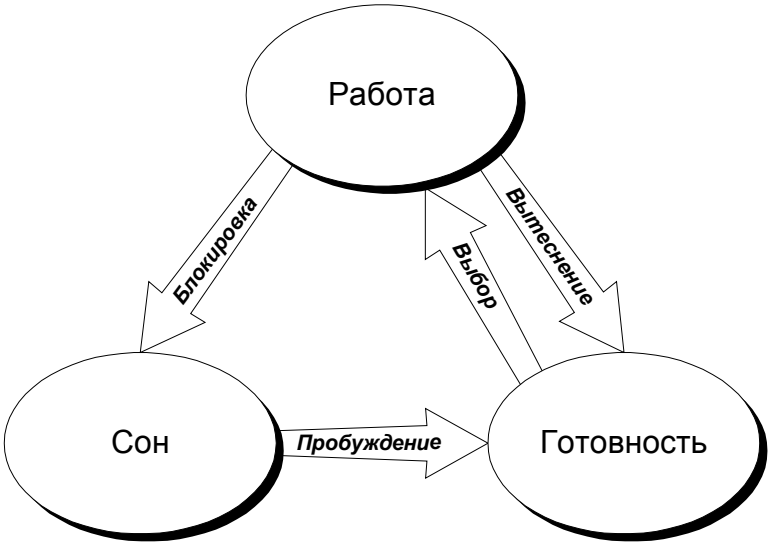


Рис. 4- 1

Розглянемо можливі переходи між станами процесу, показані на малюнку стрільцями.

Перехід **Робота → Сон** являє собою блокування процесу, що може відбутися при виклику системної функції, що блокує.

Перехід **Сон → Готовність** – це пробудження процесу, воно виконується системою при виникненні відповідної умови.

Перехід **Робота → Готовність** раніше не розглядався. Він називається ***витисненням*** процесу і виконується системою, коли вона приймає рішення про зміну поточного процесу.

Для зворотного переходу **Готовність → Робота** немає загальноприйнятого терміна. Будемо називати його ***вибором*** процесу для виконання. Відзначимо, що цей перехід майже завжди зв'язаний або з блокуванням, або з витисненням колишнього поточного процесу.

Відповісти самі на питання: чому «майже завжди», а не «завжди»? Які ще можливі варіанти?

Двох стрілок немає на діаграмі. Прямий перехід від сну до роботи нелогічний, тому що він сполучав би дві зовсім різних дії.

Яких саме?

Перехід від готовності до сну неможливий у принципі.

До речі, чому?

Крім трьох основних станів, у різних ОС можуть використовуватися й інші стани.

Стан ***старту*** означає, що процес знаходиться на етапі створення і поки не готовий вступити в роботу.

Стан ***завершення*** (у UNIX воно майже офіційно називається «зомбі») означає, що процес завершив свою роботу, але поки є присутнім у системі у виді запису про результати і причину завершення.

Стан ***припинення*** (suspended) означає, що виконання процесу тимчасово перерваний оператором (або, може бути, іншим процесом) і пізніше повинне бути їм же відновлено.

У деяких системах (наприклад, у UNIX) основні стани роздроблені на ряд більш дрібних: робота в системному й у користувальницькому режимі, готовність у пам'яті і готовність на диску і т.п. Необхідний набір станів визначається алгоритмами роботи конкретної ОС.Що

4.2.4. витісняє і невитісняє многозадачность

Усі многозадачные ОС можна розділити на два класи, що розрізняються по способах організації переключення процесів.

У системах з ***диспетчеризацією, що*** невитісняє, (non-preemptive multitasking) робота будь-якого процесу може бути перервана тільки «з ініціативи самого процесу», а точніше - тільки коли процес викликає визначені системні функції. До них відносяться, зокрема, описані вище блокують функції. Може також матися функція, спеціально призначена для добровільної поступки процесом черги на виконання (Yield). Виклик такої функції не приводить до блокування, але може привести до витиснення процесу . Якщо працюючий процес не викликає системних функцій, а займається, наприклад, довгими розрахунками, то всі інші процеси змушені простоювати. Коли ж системна функція, нарешті, викликана, то система, насамперед, перевіряє, чи може ця функція бути виконана відразу або припускає чекання деякої події (необхідний ресурс може бути зайнятий іншим процесом; операція введення/висновку звичайно вимагає визначеного часу на своє виконання; черга повідомлень, що вимагають обробки, може бути порожня). Якщо потрібне чекання, то система блокує процес, вибираючи який-небудь іншої з готових процесів для виконання. Не виключена ситуація, коли заблокованими по різних причинах виявляються всі користувальницькі процеси.

Але навіть якщо дія, необхідна процесом, може бути виконане без блокування, система не обов'язково поспішає виконати його і повернути керування поточному процесові. Насправді, момент виклику системної функції зручний для того, щоб система могла вирішити, чи не час процесові «відпочити», оскільки маються інші процеси, що давно претендують на процесорний час або обладающие більш високим пріоритетом. У цьому випадку система переводить процес у стан готовності до виконання, а один з інших готових до виконання процесів стає поточним. Що ж стосується викликаної системної функції, те її виконання буде довершено пізніше, що коли викликав процес знову стане поточної.

У попередньому п. 4.2.3 зміна поточного процесу, не зв'язана з його блокуванням, була названа витисненням процесу.

Та частина ОС, що по визначеним у системі правилам вибирає, чи варто витиснути поточний процес і який процес повинний стати наступної поточної, називається ***планувальником*** (scheduler) або ***диспетчером процесів***.

У системі з диспетчеризацією, що невитісняє, планувальник одержує керування при виклику процесом однієї з що блокують або витісняють функцій. Планувальник перевіряє, чи не виконані умови активізації одного зі сплячих процесів, а потім приймає рішення про вибір одного з активних процесів для виконання. Не виключено, що буде знову обраний той же процес (якщо тільки він не блокується).

Використання диспетчеризації, що невитісняє, дозволило розробити досить вражаючі приклади ОС, з яких найбільш відомої є Windows версій 1, 2 і 3. Звичайно в таких системах велика частина процесів знаходиться в сплячому стані, очікуючи, поки користувач звернеться до відповідного додатка. Для користувача це виглядає зовсім природно.

Головний недолік многозадачности типу, що невитісняє, полягає в тім, що будь-який процес у принципі має можливість цілком і надовго захопити процесор. Це «многозадачность на слові честі», заснована на припущенні, що програми всіх процесів написані так, щоб досить часто викликати функції, що блокують. Навіть якщо програміст пише програму для виконання складних багатогодинних обчислень, він повинний штучно вставляти в деяких місцях виклик системних функцій, що передають керування планувальникові. Якщо ж це не зроблено, то система фактично втрачає многозадачность і буде виконувати один процес, не реагуючи на дії користувача, доти, поки або «нахабний» процес завершиться, або користувач, що розлютив,

зніме його **по Ctrl+Alt+**Del.

Більш складним і зробленим типом многозадачных ОС є системи з ***диспетчеризацією*** процесів, що витісняє, (preemptive multitasking). Їхня відмінність полягає в тім, що планувальник вступає в роботу не тільки (і не стільки) при виклику системних функцій, але й у наступних двох випадках (або хоча б в одному з них):

242* коли активізується (тобто пробуджується або запускається) процес, що володіє більш високим пріоритетом, чим поточний;

243* коли минає ***квант часу***, виділений планувальником для поточного процесу.

Поняття ***пріоритету*** процесу буде докладно розглянуто трохи нижче. Пріоритет характеризує відносну важливість або терміновість даного процесу. Треба відзначити, що негайний виклик планувальника при активізації высокоприоритетного процесу характерний не для всіх систем з диспетчеризацією, що витісняє, а тільки для одного їхнього підкласу - систем ***з абсолютними*** пріоритетами. У системах з ***відносними пріоритетами*** процес з високим пріоритетом буде все-таки змушений почекати витікання кванта часу.

Системи, у яких перепланування процесу виконується після витікання кожного кванта часу, називають системами з ***квантуванням часу*** (time slicing). Якщо величина кванта досить мала, то для користувача процес періодичної зміни поточного процесу буде непомітний і створиться враження, що всі активні процеси працюють як би одночасно. З іншого боку, чим менше величина кванта, тим велику долю процесорного часу буде займати процедура переключення поточного процесу.

Виклик планувальника не обов'язково означає зміну процесу. Якщо немає інших активних процесів з таким же або більш високим пріоритетом, то планувальник може продовжити виконання того ж процесу.

Принциповою рисою систем з диспетчеризацією, що витісняє, є те, що поточний процес може бути перерваний і витиснутий практично в будь-якій крапці своєї програми. Це помітно ускладнює реалізацію таких систем у порівнянні із системами, що невитісняють, де зміна поточного процесу може відбутися тільки в момент виклику системної функції.

4.2.5. Дескриптор і контекст процесу

З кожним процесом зв'язані його дані, що описують, в основній пам'яті, необхідні ОС для підтримки виконання процесу. Усі ці дані можна розбити на дві великі структури: дескриптор процесу і контекст процесу.

Дескриптор процесу містить у собі всі ті дані про процес, що можуть знадобитися ОС при різних станах процесу. У число елементів дескриптора можуть входити, наприклад, ідентифікатор процесу (якесь умовне число, що позначає даний процес); поточний стан процесу; його пріоритет; власник процесу (тобто ідентифікатор користувача, що запустив процес); статистика витраченого процесом загального і процесорного часу; покажчик місця розташування контексту процесу й ін. Дескриптори всіх процесів, що існують у системі, зібрані в таблицю процесів.

Контекст процесу включає дані, необхідні тільки для поточного процесу. Судна відносяться, насамперед, значення всіх регістрів процесора, включаючи покажчик поточної команди; таблиця файлів, відкритих процесом; покажчики на області пам'яті, що повинний займати процес при його виконанні; значення системним перемінним, використовуваним процесом (наприклад, що текет диск і каталог, інформація про останню помилку при виконанні системних функцій); інші системні прапори і режими, що можуть мати різні значення для різних процесів.

Точний склад дескриптора і контексту сильно залежать від конкретної ОС.

При переключенні поточного процесу система повинна щораз переключати і поточний контекст, тобто зберігати у своїй пам'яті або на диску контекст попереднього процесу, що виконувався, і відновлювати раніше збережений контекст того процесу, що буде виконуватися.

4.2.6. Реентерабельность системних функцій

У многозадачной системі не можна виключити можливість того, що переключення процесів відбудеться під час виконання процесом якої-небудь з, що витісняється, системних функцій. При цьому виконання функції не буде довершено, воно буде перервано на середині. Передбачається, що виконання функції буде довершений пізніше, коли перерваний процес знову буде обраний на виконання.

Проблема полягає в тім, що новий поточний процес може викликати ту ж саму системну функцію. Виникає питання: чи можливо коректне виконання другого виклику функції, якщо до цього моменту не закінчене виконання першого виклику тієї ж функції?

Простий приклад подібної ситуації – запуск в ОС декількох програм, кожна з яких блокується на чеканні клавіатурного введення, викликавши для цього ту саму системну функцію введення.

Функція або процедура, для якої можливо коректне виконання її повторного виклику до завершення першого виклику, називається ***реентерабельної*** або ***повторно-входимой***.

Проблема реентерабельности функцій виникає в програмуванні і по зовсім іншому приводі, не зв'язаному з реалізацією ОС. Мова йде про рекурсивні функції, тобто функціях, що можуть прямо або побічно викликати самі себе. Давно відома й основна причина нереентерабельности функцій. Вона полягає у використанні тих самих комірок пам'яті для формальних параметрів і локальних перемінних при різних викликах функції. Дійсно, якщо при першому виклику функції в осередок локальної перемінної X буде занесено, наприклад, число 10, а при другому – число 20, те після поновлення виконання першого виклику значення X буде невірним.

Ліки від цієї форми нереентерабельности також давно відомо й убудовано в усі поважаючі себе мови програмування, починаючи з C і Pascal. Воно полягає в тім, що пам'ять для формальних параметрів і локальній перемінних повинна виділятися в стеці програми, при цьому кожен новий вкладений виклик одержує свій набір осередків для перемінних.

Для випадку многозадачной системи використання єдиного стека неприйнятно, оскільки виклики функцій не є вкладеними, тобто перший виклик може завершитися раніш, ніж другий, що привело б до невірного використання стека. Кожен процес одержує свій власний стек, що є частиною контексту процесу.

Ще одна причина нереентерабельности стосується тих функцій уведення/висновку, що запускають операцію і потім чекають її завершення. Повторне звертання до того ж пристрою до завершення першого виклику може привести до помилки. У даному випадку система повинна відслідковувати стан пристрою і блокувати другий виклик до звільнення пристрою.

Проблема реентерабельности системних функцій значно гостріше коштує для ОС з диспетчеризацією, що витісняє, оскільки переключення процесу може случитися при виконанні будь-якої функції. При диспетчеризації, що невитісняє, досить забезпечити реентерабельну реалізацію лише невеликого числа функцій, що блокують.

При переході від що невитісняє Windows 3.x до що витісняє Windows 95 одна із серйозних проблем складала в збереженні коду великої кількості нереентерабельних системних функцій. Проблему «вирішили» шляхом уведення семафора, що блокує повторний виклик для великого числа функцій. Неприємним наслідком цього став взаємний гальмуючий вплив процесів. У Windows NT цієї проблеми ні, усі функції реалізовані реентерабельно.

4.2.7. Дисципліни диспетчеризації і пріоритети процесів

Коли планувальник процесів одержує керування, його основною задачею є вибір наступного процесу, що повинний одержати керування. Алгоритми, що лежать в основі цього вибору, визначають ***дисципліну диспетчеризації***, прийняту в даної ОС.

Однієї із самих очевидних дисциплін є ***проста кругова диспетчеризація*** (round robin scheduling). Її суть у наступному. Всі активні процеси вважаються рівноправними й утворять кругову чергу. Кожен процес одержує від системи квант часу, після закінчення якого планувальник вибирає для виконання наступний процес з черги. Таким чином, якщо всі процеси залишаються активними, то система забезпечує їхнє рівномірне просування, що імітує рівнобіжне виконання всіх процесів. Якщо поточний процес блокується, він випадає з кола і попадає в список сплячих процесів. Коли система активізує один зі сплячих процесів, він включається в кругову чергу.

У деякому змісті протилежною дисципліною є ***фоново-оперативная диспетчеризація*** (foreground/background scheduling) – одна із самих старих форм організації многозадачной роботи. У найпростішому випадку вона включає два процеси: фоновий процес і оперативний процес (процес переднього плану). Фоновий процес виконується тільки тоді, коли спить оперативний процес. При активізації оперативного процесу відбувається негайне витиснення фонового, тобто оперативний процес має більш високий абсолютний пріоритет. Звичайно при такій дисципліні передбачається, що активізація оперативного процесу не зажадає багато процесорного часу, так що виконання фонового процесу буде незабаром відновлено. Одним із прикладів ефективного використання фоново-оперативной диспетчеризації є так називана «фонова печатка», що дозволяє виконати навіть однозадачная MS-DOS. При цьому процес висновку файлу на принтер розглядається як процес переднього плану, а звичайна діалогова робота з ОС – як фоновий процес. Оскільки обслуговування переривань від принтера займає лишь частки відсотка процесорного часу, користувач не відчуває ніякого уповільнення роботи.

Між описаними двома крайностями лежить велика розмаїтість дисциплін ***пріоритетної диспетчеризації***. Усі вони засновані на приписуванні кожному процесові при його створенні деякого числа – ***пріоритету***. Більш високий пріоритет повинний давати процесові визначені переваги перед низкоприоритетными процесами при роботі планувальника.

Призначення пріоритетів виконується користувачем або адміністратором системи, можливо також програмна зміна пріоритету процесу. На вибір оптимального рівня пріоритету впливають в основному два розуміння:

244* важливість, відповідальність даного процесу або привілейоване положення процес користувача, що запускає;

245* кількість процесорного часу, на которое буде претендувати процес (як ми бачили в прикладі з фоновою печаткою, високий пріоритет процесу, що мало завантажує процесор, майже не приводить до уповільнення роботи інших процесів).

Основний алгоритм пріоритетного планування нагадує просте кругове планування, однак кругова черга активних процесів формується окремо для кожного рівня пріоритету. Поки є хоч один активний процес у черзі з найвищим пріоритетом, процеси з більш низькими пріоритетами не можуть одержати керування. Тільки коли всі процеси з вищим пріоритетом заблоковані або довершені, планувальник вибирає процес з черги з більш низьким пріоритетом.

Пріоритет, що привласнюється процесові при створенні, називається ***статичним*** пріоритетом. Дисципліна планування, що використовує тільки статичні пріоритети, має один істотний недолік: низкоприоритетные процеси можуть надовго виявитися цілком відлученими від процесора. Іноді це прийнятно (якщо высокоприоритетные процеси незрівнянно важливіше, ніж низкоприоритетные), однак частіше хотілося б, щоб і на низькі пріоритети хоч що-небудь

перепадало, нехай навіть рідше й у меншій кількості, чим на високі. Для рішення цієї задачі запропонована безліч різних алгоритмів планування процесів, заснованих на ідеї ***динамічного*** пріоритету.

Динамічний пріоритет процесу – це величина, розраховува автоматично системою на основі двох основних факторів: статичного пріоритету і ступеня попереднього використання процесора даним процесом. Загальна ідея наступна: якщо процес занадто довго не одержував процесорного часу, те його пріоритет варто підвищити, щоб дати процесові шанс на майбутнє. Навпаки, якщо процес занадто часто і довго працював, їсти зміст тимчасово понизити його пріоритет, щоб пропустити вперед зголоднілих конкурентів.

Можуть враховуватися й інші розуміння, що впливають на динамічний пріоритет. Наприклад, якщо процес веде діалог з користувачем, то має сенс підвищити його пріоритет, щоб скоротити час реакції й уникнути прикрих затримок при натисканні клавіш. Якщо процес останнім часом часто блокувався, не використовувавши до кінця виділений йому квант часу, то це теж підстава для підвищення пріоритету: цілком можливо, процес і надалі буде так само невибагливий.

4.3. Проблеми взаємодії процесів

4.3.1. Ізоляція процесів і їхня взаємодія

Одна з найважливіших цілей, що ставляться при розробці многозадачных систем, полягає в тім, щоб різні процеси, що одночасно працюють у системі, були якнайкраще ізольовані друг від друга. Це означає, що процеси (в ідеалі) не повинні нічого знати навіть про існування один одного. Для кожного процесу ОС надає ***віртуальну машину***, тобто повний набір ресурсів, що імітує виконання процесу на окремому комп'ютері.

Ізоляція процесів, по-перше, є необхідною умовою надійності і безпеки многозадачной системи. Один процес не повинний мати можливості втрутитися в роботу іншого або одержати доступ до його даних, ні по випадковій помилці, ні навмисно.

По-друге, проектування і налагодження програм надзвичайно ускладнилися б, якби програміст повинний був враховувати непередбачений вплив інших процесів.

З іншого боку, є ситуації, коли взаємодія необхідна. Процеси можуть спільно обробляти загальні дані, обмінюватися повідомленнями, чекати відповіді і т.п. Система повинна надавати в розпорядження процесів засобу взаємодії. Це не суперечить тому, що вище було сказано про ізоляції процесів. Щоб взаємодія не привела до повного хаосу, воно повиннео виконуватися тільки за допомогою тих добре продуманих засобів, що надає процесам ОС. За межами цих засобів діє ізоляція процесів.

Це як границя держав – перетинати неї в довільних місцях заборонено, але повинна бути добре обладнана система пропускних пунктів, де легко проконтролювати дотримання правил перетинання границі.

Поняття взаємодії процесів містить у собі кілька видів взаємодії, основними з яких є:

246* синхронізація процесів, тобто, спрощено говорячи, чекання одним процесом яких-небудь подій, зв'язаних з роботою інших процесів;

247* обмін даними між процесами.

Набір засобів, призначених для взаємодії процесів, часто позначають абревіатурою IPC (InterProcess Communication). Склад функцій і методів обумовлений багаторічним досвідом як програмістів-практиків, так і теоретиків, що розглядає проблеми взаємодії рівнобіжних процесів.

Випливає ще раз відзначити, що проблеми взаємодії процесів не залежать від способу

реалізації рівнобіжної роботи процесів у конкретної ОС. Велика частина цих проблем має місце в однаковій мірі і для квазипараллельной реалізації, і для щирої паралельності з використанням декількох процесорів. Засобу рішення проблем, щоправда, можуть залежати від реалізації.

4.3.2. Проблема взаємного виключення процесів

Серйозна проблема виникає в ситуації, коли два (або більш) процеси одночасно намагаються працювати з загальними для них даними, причому хоча б один процес змінює значення цих даних.

Проблему часто пояснюють на такому, небагато умовному, прикладі. Нехай мається система резервування авіаквитків, у якій одночасно працюють два процеси. Процес А забезпечує продаж квитків, процес В – повернення квитків. Не поглиблюючи в деталі, будемо вважати, що обидва процеси працюють з перемінної **N** – числом квитків, що залишилися, причому відповідні фрагменти програм на псевдокоді виглядають приблизно так:

Процес А:	***Процес В:***
`. . .`	`. . .`
`R1 := N;`	`R2 := N;`
`R1 := R1 - 1;`	`R2 := R2 + 1;`
`N := R1;`	`N := R2;`
`. . .`	`. . .`

Уявимо собі тепер, що при квазипараллельной реалізації процесів у ході виконання цих трьох операторів відбувається переключення процесів. У результаті, у залежності від непередбачених випадків, порядок виконання операторів може виявитися різним, наприклад:

1) `R1 := N; R2 := N; R2 := R2 + 1; N := R2; R1 := R1 - 1; N := R1;`
2) `R2 := N; R2 := R2 + 1; R1 := N; R1 := R1 - 1; N := R1; N := R2;`
3) `R1 := N; R1 := R1 - 1; N := R1; R2 := N; R2 := R2 + 1; N := R2;`

Ну і що? А те, що у випадку 1 значення **N** у результаті виявиться зменшеним на 1, у випадку 2 – збільшеним на 1, і тільки у випадку 3 значення **N**, як і покладено, не зміниться.

Можна привести менш екзотичні приклади.

248* Якщо два процеси одночасно намагаються відредагувати ту саму запис бази даних, то в результаті різні поля однієї записи можуть виявитися неузгодженими.

249* Якщо один процес додає повідомлення в чергу, а іншої в цей час намагається взяти повідомлення з черги для обробки, то він може прочитати не цілком сформоване повідомлення.

250* Якщо два процеси одночасно намагаються звернутися до диска, кожний зі своїм запитом, то що саме кожний з них у результаті прочитає або запише на диск – сказати важко.

Ситуація зрозуміла: не можна дозволяти двом процесам одночасно звертатися до тим самим даних, якщо при цьому відбувається зміна цих даних.

Те, що ми розглядали квазипараллельную реалізацію процесів, не настільки істотно. Для процесів, що працюють на різних процесорах, але одночасно звертаються до тим самим даних, ситуація приблизно та ж.

Задовго до створення многозадачных систем розроблювачі засобів автоматики зштовхнулися з неприємним ефектом залежності результату операції від випадкового і непередбаченого співвідношення швидкостей поширення різних сигналів в електронних схемах. Цей ефект вони назвали «гонками». Ми тут ведемо мову, по суті, про тім же.

Для більш чіткого опису ситуації було введене поняття ***критичної секції***.

Критичною секцією процесу стосовно деякого ресурсу називається така ділянка програми процесу, при проходженні якого необхідно, щоб ніякий інший процес не знаходився у *своїй* критичній секції стосовно *того ж* ресурсу.

У прикладі з квитками приведені три оператори в кожнім із процесів складають критичну секцію цього процесу стосовно загальної перемінного **N**. Алгоритм роботи кожного процесу окремо правильний, але правильна робота двох процесів у сукупності може бути гарантована, тільки якщо вони не сунуться одночасно кожний у свою критичну секцію.

А як їм це заборонити?

На перший погляд здається, що ця проблема (її називають проблемою ***взаємного виключення*** процесів) вирішується просто. Наприклад, можна ввести булеву перемінну **Free**, доступним обом процесам і имеющую зміст «критична область вільна». Кожен процес перед входом у свою критичну область повинний очікувати, поки ця перемінна не стане щирої, як показано нижче:

Процес A:	***Процес B:***
`. . .`	`. . .`
`while not Free do`	`while not Free do`
`  ;`	`  ;`
`Free := false;`	`Free := false;`
`(критична секція A)`	`(критична секція B)`
`Free := true;`	`Free := true;`
`. . .`	`. . .`

В обох процесах цикл **while** не робить нічого, крім чекання, поки інший процес вийде зі своєї критичної секції.

А не потрібно чи було що-небудь зробити з перемінної **Free** ще до запуску процесів A і B?

Насамперед, відзначимо, що запропоноване рішення використовує таку неприємну річ, як активне чекання: процесорний час розтрачується на багаторазову перевірку перемінної **Free**. Але це полбеды.

Лихо в тім, що таке рішення нічого не вирішує. Якщо реалізувати його на практиці, то «неприємності» стануть рідше, але не зникнуть. У першому, нехитрому варіанті програми угрожаемыми ділянками були критичні секції обох процесів. Тепер же уразлива ділянка звузилася до однієї крапки, відзначеної в програмі кожного процесу штриховою лінією. Це крапка між перевіркою перемінної **Free** і зміною цієї перемінної. Якщо переключення процесів відбудеться, процес, що коли витісняється, буде знаходитися саме в цій крапці, то спочатку в критичну секцію ввійде (з повним правом на це) інший процес, а потім, коли керування повернеться до першого процесу, він без додаткової перевірки теж ввійде у свою критичну секцію.

Розроблювачі перших програмних систем, що використовують взаємодію рівнобіжних процесів, не відразу усвідомили складність проблеми взаємного виключення. Були випробувані різні програмні рішення, перш ніж удалося знайти таке, котре задовольняло трьом природним умовам:

251* у будь-який момент часу не більш, ніж один процес може знаходитися в критичній секції;

252* якщо критична секція вільна, то процес може безперешкодно ввійти в неї;

253* усі процеси рівноправні.

Спробуйте самі знайти таке рішення. У книгах /3/ і /4/ можна знайти кілька варіантів рішення з аналізом їхніх помилок.

Зрештою правильні алгоритми вирішений задачі взаємного виключення запропонували спочатку Деккер (його алгоритм досить заплутаний, що часто трапляється з першими рішеннями

складних задач), потім Питерсон, чий алгоритм простіше і зрозуміліше.

Для допитливих приводимо рішення Питерсона. У ньому використовуються булевы перемінні **flag**, **flag**, споконвічно рівні **false**, і перемінна типу, що **перелічується, turn: A..B**.

Процес A:

```
. . .
flagA := true;
turn := B;
while flagB and turn = B do
  ;
(критична секція A)
flagA := false;
. . .
```

Процес B:

```
. . .
flagB := true;
turn := A;
while flagA and turn = A do
  ;
(критична секція B)
flagB := false;
. . .
```

Приведений алгоритм дійсно вирішує проблему, однак у нього є два істотних недоліки. По-перше, якщо в конкуренції за критичну секцію беруть участь не два процеси, а три або більш, те програма стає дуже громіздкою. По-друге, рішення Питерсона засноване на використанні активного чекання.

Ще одним напрямком у реалізації взаємного виключення стало включення спеціальних машинних команд у набори команд нових процесорів. Оскільки небезпека, як ми бачили, зв'язана з поділом за часом операцій перевірки і присвоювання, те були запропоновані команди, що виконують одночасно перевірку і присвоювання. Такі команди є, наприклад, у процесорів Pentium. З їхньою допомогою дійсно можна простіше реалізувати взаємне виключення, але для цього все рівно потрібне активне чекання.

4.3.3. Двоичные семафори Дейкстры

Зовсім іншим способом підійшов до проблеми взаємного виключення великий голландський учений Э.Дейкстра (E.Dijkstra, 1966). Він запропонував використовувати новий вид програмних об'єктів – ***семафори***. Тут ми розглянемо їхній найпростіший варіант – ***двоичные семафори***, вони ж ***мьютексы*** (mutex, від слів MUTual EXclusion – взаємне виключення).

Двоичным семафором називається перемінна S, що може приймати значення 0 і 1 і для якої визначені тільки дві операції.

254* P(S) – операція заняття (закриття) семафора. Вона очікує, поки значення S не стане рівним 1, і, як тільки це случиться, привласнює S значення 0 і завершує своє виконання. *Дуже важливо*: операція P *по визначенню* неподільна, тобто між перевіркою і присвоюванням не може вклинитися інший процес, який би змінив значення S.

255* V(S) – операція звільнення (відкриття) семафора. Вона просто привласнює S значення 0.

Чим перемінн-семафор відрізняється від звичайної булевой перемінної? Тим, що для неї неприпустимі ніякі інші операції, крім P і V. Не можна написати в програмі **S:=1** або **if(S)then ...** , якщо **S** визначена як семафор.

Чим операція P відрізняється від варіанта з перевіркою і присвоюванням, що ми вище визнали незадовільним? Неподільністю. Але це «по визначенню», а як на практиці домогтися цієї неподільності? Це окремий, цілком розв'язуваний питання.

Заслуга Дейкстры саме в тім, що він розділив проблему взаємного виключення на дві незалежні проблеми різних рівнів:

256* на рівні реалізації: як забезпечити роботу семафорів відповідно до їх визначення;

257* на рівні взаємодії процесів: як написати коректно працюючу програму, якщо в розпорядженні програміста маються семафори.

Вирішувати ці дві задачі по окремості легше, ніж обидві разом, при цьому вирішувати їх звичайно повинні різні люди: першу – розроблювачі ОС, а другу – розроблювачі прикладної програми.

Розглянемо спочатку реалізацію. Очевидно, функції P і V зручніше і надійніше один раз реалізувати в ОС, чим щораз по-новому – у прикладних програмах. (Назви цих функцій можуть у конкретних системах бути й іншими, більш виразними.)

Системна функція P(S) повинна перевірити, чи вільний семафор S. Якщо вільно (S = 1), то система займає його (S := 0) і на цьому функція завершується. Якщо ж семафор зайнятий, то система блокує процес, що викликав функцію P, і запам'ятовує, що цей процес блокований по чеканню звільнення семафора S. Таким чином, при реалізації семафорів вдається уникнути активного чекання.

Неподільність операції забезпечується тим, що під час виконання системою функції P переключення процесів заборонене. У крайньому випадку, ОС має можливість для цього на короткий час заборонити переривання.

Системна функція V(S) – це, звичайно, не проста присвоювання S := 1. Крім цього, система повинна перевірити, немає чи серед сплячих процесів такого, котрий очікує звільнення семафора S. Якщо такий процес найдеться, система розблокує його, а перемінна S у цьому випадку зберігає значення 0 (семафор знову зайнятий, тепер вже іншим процесом).

Чи може случитися так, що кілька сплячих процесів чекають звільнення того самого семафора? Так, так цілком може бути. Який з цих процесів повинний бути розбуджений системою? З погляду коректності роботи і відповідності визначенням функцій P і V – будь-який, але тільки один. З погляду ефективності роботи – імовірно, треба розбудити самий пріоритетний процес, а у випадку рівності пріоритетів…ну, видимо, той, котрий спить довше.

Тепер, коли ми розібралися з реалізацією семафорів, можна про неї забути і пам'ятати тільки, що семафори існують і можуть бути використані при необхідності.

Розглянемо тепер другу половину задачі – використання семафорів для керування взаємодією процесів. Як можна реалізувати коректну роботу процесів із критичними секціями, якщо використовувати двоичный семафор? Так дуже просто.

Процес A:	***Процес B:***
`. . .`	`. . .`
`P(S);`	`P(S);`
`(критична секція A)`	`(критична секція B)`
`V(S);`	`V(S);`
`. . .`	`. . .`

І усі. Складності пішли в реалізацію семафорів. Треба тільки простежити, щоб до початку роботи процесів семафор S був відкритий.

4.3.4. Засобу взаємодії процесів

Можна довести, що використання двоичных семафорів дозволяє коректно вирішити будь-як проблеми синхронізації процесів. Але зовсім не обов'язково це рішення виявиться простим і зручним. У деяких випадках використання семафорів повинне все-таки супроводжуватися небажаним активним чеканням.

За десятиліття, що пройшли після винаходу семафорів, були запропоновані різні засоби синхронізації, більш пристосовані для різних типових задач. Розглянемо деякі з них.

4.3.4.1. Целочисленные семафори

У згаданій роботі Дейкстры, крім двоичных семафорів, що приймають значення 0 і 1, був розглянутий також більш загальний тип семафорів зі значеннями на інтервалі від 0 до деякого N. Функція P(S) зменшує позитивне значення семафора на 1, а при нульовому значенні переходить у чекання, як і у випадку двоичного семафора. Функція V(S) збільшує значення семафора на 1, але не більш N.

Область застосування целочисленных семафорів трохи інша, чим у двоичных. Целочисленные семафори застосовуються в задачах виділення ресурсів з обмеженого запасу. Величина N характеризує загальну кількість наявних одиниць ресурсу, а поточне значення перемінної – кількість вільних одиниць. При запиті ресурсу процес викликає функцію V(S), при звільненні – P(S).

Для целочисленных семафорів іноді зручно використовувати модифіковану функцію V(S, k), другим параметром якої є число одночасне запитуваних одиниць ресурсу. Така функція блокує процес, якщо значення семафора менше k.

4.3.4.2. Семафори з множинним чеканням

Можлива ситуація, коли процес може вибрати один з декількох шляхів подальшої роботи, але на кожнім шляху він може бути заблокований закритим семафором. Розумно було б чекати звільнення кожного із семафорів і тільки тоді вибрати вільний шлях. Але як це зробити? Викликавши P(S) для одного із семафорів, процес приречений чекати звільнення саме цього семафора, а не кожного з наявних.

Життєва ситуація: покупець у супермаркеті, що вибирає, до якій з кас зайняти черга. Добре б угадати черга, що пройде швидше…

Функція ***множинного чекання*** $P(S_1, S_2, \ldots S_n)$ дозволяє вказати як параметри кілька двоичных семафорів (або масив семафорів). Якщо хоча б один із семафорів вільний, функція займає його, у противному випадку вона чекає освобождения кожного із семафорів.

Інший, не менш корисний варіант множинного чекання, це чекання моменту, коли *всі* зазначені семафори виявляться вільні. Це означає, що процес може працювати далі тільки в тому випадку, якщо одночасно виконані кілька умов, кожне з яких задане у виді двоичного семафора.

4.3.4.3. Сигнали

Сигнал – це щось, що може бути послано процесові системою або іншим процесом. Із сигналом не зв'язано ніякої інформації, крім номера (коду), що вказує, який саме тип сигналу посилається. При одержанні сигналу процес перериває свою поточну роботу і переходить на виконання функції, визначеної як оброблювач сигналів даного типу.

Таким чином, сигнали сильно схожі на переривання, але тільки высокоуровневые, керовані системою, а не апаратурою.

Механізм сигналів дозволяє вирішити, наприклад, проблему критичної секції іншим способом, чим семафори.

Подумайте самостійно, як це можна зробити.

4.3.4.4. Повідомлення

Повідомлення також посилаються процесові системою або іншим процесом, однак відрізняються від сигналів у двох відносинах.

По-перше, повідомлення не переривають роботу процесу-одержувача. Замість цього вони стають у чергу повідомлень. Процес повинний сам викликати функцію прийому повідомлення. Якщо черга порожня, ця функція блокує процес до одержання якого-небудь повідомлення.

По-друге, з повідомленням, на відміну від сигналу, може бути зв'язана інформація,

передана одержувачеві. Таким чином, повідомлення – це засіб не тільки синхронізації, але й обміну даними між процесами.

4.3.4.5. Загальна пам'ять

Поговоримо тепер ще про обмін даними. Найпростішим і природним способом такого обміну представляється можливість спільного доступу двох або більш процесів до загальної області пам'яті. Але оскільки звичайно ОС прагне, навпаки, надійно розділити пам'ять різних процесів, то для виділення обший пам'яті потрібні спеціальні системні засоби.

Загальна пам'ять служить тільки засобом обміну даними, але ніяк не вирішує проблем синхронізації. Ділянки програми, де відбувається робота з загальною пам'яттю, часто варто розглядати як критичні секції і захищати семафорами.

4.3.4.6. Програмні канали

Інше часто використовуваний засіб обміну даними – програмний канал (pipe; іноді переводиться як «трубопровід»). У цьому випадку для виконання обміну використовуються не команди читання/запису в пам'ять, а функції читання/запису у файл. Програмний канал «прикидається файлом», для роботи з ним використовуються ті ж операції, що для послідовного доступу до файлу: відкриття, читання, запис, закриття. Однак джерелом даних, що читається, служить не файл на диску, а процес, що виконує запис «в інший кінець труби». Дані, записаним одним процесом, але поки не прочитані іншим, зберігаються в системному буфері. Якщо ж процес намагається прочитати дані, що поки не записані іншим процесом, то процес-читач блокується до одержання даних.

4.3.5. Проблема тупиків

Відповідно до визначення з /7/, тупик – це стан, у якому «*деякі процеси заблоковані в результаті таких запитів на ресурси, що* ніколи *не можуть бути задоволені, якщо не будуть початі* надзвичайні *системні міри*».

Як це накажете розуміти?

Насамперед, давайте відзначимо, що процесові, що діє поодинці, не під силам загнати пристойну ОС у тупик. Вимоги процесу не будуть задоволені, тільки якщо вони перевищують те, що є в системи. Скажемо, процес вимагає 500 Мб оперативної пам'яті, коли в системи є той^-те-вес-те 256 Мб. Ну, так у цьому випадку процес буде не блокований, а нещадно убитий системою.

Інша справа, якщо в справі замішані два або більш процеси. Відповідно до іншого визначення, даному в /2/, «*Група процесів знаходиться в тупиковій ситуації, якщо кожен процес із групи очікує події, що може викликати только інш процес з тієї ж групи*».

Розглянемо такий приклад. Нехай кожний із процесів A і B збирається працювати з двома файлами, F1 і F2, причому не має наміру розділяти ці файли з іншим процесом. Програми ж процесів злегка розрізняються, а саме:

Процес A:	***Процес B:***
`. . .`	`. . .`
`Відкрити(F1);`	`Відкрити(F2);`
`Відкрити(F2);`	`Відкрити(F1);`
`(робота процесу A з файлами);`	`(робота процесу B з файлами);`
`Закрити(F1);`	`Закрити(F1);`
`Закрити(F2);`	`Закрити(F2);`
`. . .`	`. . .`

У цій ситуації усіх може пройти благополучно. Нехай, наприклад, процес A устигне

відкрити обидва файли до того моменту, коли процес В спробує відкрити F2. Ця спроба не увінчається успіхом: процес В або буде заблокований до звільнення файлів, або одержить повідомлення про помилку при відкритті файлу і, якщо процес розумний, через якийсь час спробує ще раз. Зрештою обидва процеси одержать необхідні ресурси (у даному випадку відкриті файли), хоча і не обоє відразу.

Зовсім інше буде справа, якщо А устигне відкрити тільки F1, після чого В відкриє F2. Те^-те-отут-те і вийде тупик. Процес А хоче відкрити файл F2, але не зможе цього зробити раніш, ніж В закриє цей файл. Але В не закриє F2 до того, як зуміє відкрити файл F1, що зайнятий процесом А. Кожний із процесів захопив один з ресурсів і не збирається його віддавати раніш, ніж одержить іншої. Ситуація «двох баранів на мосту».

Підкреслимо: тупик – це не просто блокування процесу, коли необхідний йому ресурс зайнятий. Зайнятий – ну і що, згодом, либонь, звільниться. Тупик – це ***взаємне блокування***, з якого немає виходу.

Ще приклад. Нехай у системі мається 100 Мб пам'яті, доступної для процесів. Процес А при своєму старті займає 40 Мб, але пізніше на короткий час вимагає ще 30 Мб, після чого завершується, звільняючи всю пам'ять. Процес В поводиться точно в такий же спосіб.

Кожний із процесів по окремості не вимагає в системи нічого неможливого. У той же час зрозуміло: якщо обидва процеси зуміють стартувати і почати рівнобіжну роботу, те жоден з них ніколи не одержить додаткові 30 Мб. Тупик.

Зовсім грубий приклад. Процес А на якомусь етапі роботи чекає повідомлення від процесу В, після чого збирається послати відповідне повідомлення. У той же час процес В чекає повідомлення від А, щоб потім відповісти на нього. Тупик неминучий. У даному випадку, на відміну від попередніх, виникнення тупика зв'язане з явною помилкою в логіку програми.

Чи може допомогти в боротьбі з тупиками використання семафорів і інших подібних засобів? Навряд чи, хіба що в рідких випадках. Семафор – це, скоріше, додаткове гальмо для процесу. Річ корисна, але не сприятливому просуванню.

Усі відомі способи боротьби з тупиками можна розділити на три групи:

258* виключення можливості тупиків шляхом аналізу вихідного тексту програм;

259* запобігання виникнення тупиків при роботі ОС;

260* ліквідація виниклих тупиків.

Що стосується аналізу тексту – це, безумовно, потрібна річ, хоча і непроста. Визначити по тексту програм процесів, чи можуть вони зайти в тупик – складна задача. До того ж, якщо і можуть, те зовсім не обов'язково зайдуть, усіх може залежати від конкретних вихідних даних і від тимчасових співвідношень. Але головне – для аналізу вихідного тексту програм потрібно мати у своєму розпорядженні цей текст. Чи реально це? Тільки в деяких ситуаціях. Наприклад, при розробці убудованої системи вихідні тексти всіх прикладних програм звичайно доступні розроблювачеві ОС. Звичайно, у цьому випадку аналіз на можливість тупиків просто необхідний. Інший приклад – розробка складного многопроцессного додатка, коли розроблювач повинний хоча б виявити можливість взаємного блокування між «своїми» процесами.

Якщо тексти недоступні, то можна спробувати запобігати тупики вже в ході роботи програм, відслідковуючи їхні запити на ресурси і блокуючи або припиняючи тим процесам, що, очевидно, «лізуть у тупик».

Самий грубий з подібних підходів полягає в тім, що кожен процес повинний захоплювати всі необхідні йому ресурси відразу, при старті, після чого він може або утримувати ресурси протягом усього часу роботи, або звільняти ресурси, що більше не потрібні. Такий підхід, безумовно, у корені виключає можливість тупиків. На жаль, на практиці він майже виключає і

многозадачную роботу: багато ресурсів необхідні для роботи більшості процесів (наприклад, диски, принтер, системні файли), тому поки один процес утримує всі ресурси, інші не зможуть навіть стартувати.

Ледве получше ***алгоритм нумерованих ресурсів***. Він полягає в тім, що всі ресурси, що маються в системі, нумеруються цілими числами в довільному порядку (хоча, імовірно, для підвищення ефективності найкраще пронумерувати них у порядку зростання дефіцитності ресурсу). Далі застосовується простої правило: запит процесу на виділення йому ресурсу з номером K задовольняється тільки в тому випадку, якщо процес у даний момент не володіє ніяким іншим ресурсом з номером N (K. Іншими словами, запит ресурсів варто виконувати тільки в порядку зростання номерів. Неважко показати, що це правило є достатньою умовою відсутності тупиків. Але ця умова занадто обмежуюче, вона відтинає багато ситуацій, коли тупик насправді не виник би.

Найбільш витончений ***алгоритм банкіра***, запропонований тим же Дейкстрой. У ньому передбачається, що кожен процес при старті повинний оголосити системі, на які ресурси й у якій максимальній кількості він може в майбутньому претендувати. Далі, уводиться поняття ***безпечного*** (у відношенні тупиків) ***стану системи***. Поточний стан, у якому мається набір процесів, кожний з яких володіє деякими ресурсами, вважається безпечним, якщо наявних у наявності вільних ресурсів досить для того, щоб ОС змогла у визначеній послідовності задовольнити максимальні запити кожного процесу.

Це визначення простіше зрозуміти, якщо розглянути, як перевіряється безпека заданого стану. Відповідно до алгоритму, серед наявних процесів шукається такий, чиї максимальні заявлені запити система може задовольнити, використовуючи наявні вільні ресурси. Якщо хоча б один такий процес знайдений, то він викреслюється з загального списку і всі ресурси, що він уже встиг зайняти, вважаються вільними (тобто вважається, що система змогла завершити цей процес). Далі, при вільних ресурсах, що збільшилися, шукається наступний процес, чиї запити система може задовольнити, і т.д. Якщо в результаті удається викреслити всі процеси, то аналізований стан системи є безпечним.

Сам же алгоритм банкіра можна тепер сформулювати дуже просто: будь-який запит процесу на виділення йому додаткових ресурсів повинний задовольнятися тільки в тому випадку, якщо стан, у яке перейде система після цього виділення, буде безпечним.

Назва алгоритму навіяна, видимо, образом обережного банкіра, що видає кредит тільки в тому випадку, якщо після цього зможе виконати усі свої зобов'язання навіть у гіршому випадку. Явно не російський банкір.

Хоча даний алгоритм красивий і логічний, у нього є, щонайменше, два недоліки. По-перше, у сучасних ОС не прийнято жадати від процесів, щоб ті заздалегідь повідомляли свої майбутні потреби в ресурсах. По-друге, виконувати перевірку безпеки стану при кожнім запиті кожного з процесів – занадто трудомістке задоволення.

Розглянемо, нарешті, третій підхід – ліквідацію уже виниклих тупиків, без спроб запобігти їхнє виникнення. У книзі /2/ цей підхід названий «алгоритмом страуса».

Тут, насамперед, виникає питання: як переконатися, що система дійсно в тупику? Зовнішньою ознакою цього є тривала відсутність якої-небудь активності двох або більш процесів. Але це недостовірна ознака, процеси могли просто надовго задуматися над яким-небудь трудомістким обчисленням. Є алгоритми, що аналізують поточний стан процесів і ресурсів, наявність заблокованих запитів, і на цій основі ставлять діагноз тупика. У принципі, такий алгоритм міг би бути убудований в ОС. Однак у літературі немає зведень про те, щоб це було здійснено на практиці. Звичайно покладаються на вольове рішення оператора або адміністратора системи.

Але нехай навіть точно відомо, що тупик є. Як можна його усунути? Як «розтаскати

баранів з моста»? Як правило, для цього застосовується радикальне рішення: примусово припинити один з тупикових процесів (скинути одного барана в ріку). Якщо не допомогло – утопити наступного барана. І т.д.

У літературі розглядаються і більш гуманні, але складні способи. У принципі, можна регулярно запам'ятовувати стан всіх або тільки найбільш відповідальних процесів у визначених контрольних крапках або через визначені періоди часу. Тоді, замість повного припинення процесу, його можна повернути до однієї з останніх контрольних крапок і притримати там, поки інший баран не перейде через ріку.

Завершуючи розгляд проблеми тупиків, варто визнати, що в даний час її практичне значення значне менше, ніж хотілося б теоретикам. По-перше, зовсім не легко загнати в тупик сучасну ОС, що працює на комп'ютері з величезними ресурсами. У прикладі ми розглянули тупик, що виник через 100 Мб пам'ять, але якщо врахувати ще трохи гигабайт, які можна використовувати у файлі підкачування, то запити процесів повинні бути вуж дуже великі, щоб привести до тупика. А, скажемо, такий пристрій, як принтер, у сучасних ОС узагалі не може стати причиною тупика, тому що система не віддає його у володіння жодному процесові навіть на час.

По-друге, хоча тупик у принципі залишається можливим, користувач навряд чи навіть помітить його. Скоріше, він скаже «Знову Windows зависла!» і перезавантажить систему.

Запобігання тупиків залишається дійсно важливим, наприклад, при розробці убудованої системи, що повинна тривалий час безвідмовно працювати без утручання людини.

4.4. Керування процесами в MS-DOS

4.4.1. Процеси в MS-DOS

Як говорилося вище, керування процесами в однозадачных ОС, до яких відноситься MS-DOS, є порівняно тривіальною задачею.

Завантаження ОС завершується запуском програми командного інтерпретатора COMMAND.COM, у задачі якого входить:

261* читання й аналіз команд, що вводяться користувачем із клавіатури;

262* виконання внутрішніх команд системи, таких, як команда видачі змісту каталогу, команди копіювання, видалення і перейменування файлів і т.п.;

263* запуск на виконання системних і прикладних програм;

264* обробка критичних помилок, що происшли в ході виконання системних функцій MS-DOS;

265* завершення роботи програми зі звільненням усіх ресурсів, що займалися програмою.

Запускаючи програму користувача, COMMAND.COM не завершує власну роботу, а фактично переходить у стан сну. Після завершення запущеної програми COMMAND.COM відновляє роботу, видаючи запрошення до введення наступної команди. У такий же спосіб програма користувача може запустити іншу програму й очікувати її завершення. Кількість одночасна присутніх у системі процесів обмежено тільки розміром пам'яті системи (не більш 640 Кб на усіх), однак тільки остання запущена програма може бути в активному (працюючому) стані. Якщо ж і ця програма блокується на виконанні системної функції (наприклад, очікує введення з клавіатури), то в системі не залишається активних процесів. Таким чином, термін «однозадачная ОС» у даному випадку варто розуміти як «ОС, що допускає не більш однієї *активної* задачі». Системі не приходиться займатися поділом процесорного часу й іншими «многозадачными» проблемами, за винятком тільки збереження і відновлення контексту

батьківської програми.

Деяким виключенням із правила «одна активна задача» є резидентные програми MS-DOS, розглянуті в п. 4.4.5.

4.4.2. Середовище програми

Середовище програми (environment; інший переклад – «оточення») являє собою текстовий масив, що складається з рядків виду:

"*перемінна=значення*", 0

Тут *перемінна* і *значення* – будь-які (у розумних межах) текстові величини, байт 0 завершує кожен рядок.

Поняття середовища було введено в системі UNIX і запозичено відтіля в MS-DOS і Windows без особливих змін.

Мається трохи стандартних (системних) перемінні середовища, з яких найбільш відомі **PATH** (визначає шляхи до каталогів, у яких система шукає файл, що виконується,) **i** PROMPT (задає вид підказки при діалозі з ОС). Крім того, багато прикладних програм вимагають для правильної роботи, щоб були задані специфічні перемінні середовища, що описують, наприклад, розміщення робочих каталогів програми, спосіб роботи з розширеною пам'яттю або якісь інші характеристики режиму роботи програми.

Можна розглядати перемінні середовища як свого роду параметри, передані програмі при її запуску, аналогічно тому, як підпрограма одержує параметри при виклику. Інтерпретатор команд COMMAND.COM також має своє середовище, що називають кореневим середовищем. Для створення перемінних кореневий середовища, їхнього видалення і зміни значень може використовуватися системна команда **SET**. Коли COMMAND.COM запускає програму користувача або одна програма запускає іншу, створюється породжений процес, що одержує власний екземпляр блоку середовища, при цьому за замовчуванням створюється точна копія середовища батька, однак можна створити зовсім інше середовище.

4.4.3. Запуск програми

Однієї з основних задач, що повинна вирішувати система, є запуск програм на виконання. Для цього призначена системна функція **Exec**, що може бути викликана або програмою COMMAND.COM, що виконує команду користувача, або безпосередньо програмою користувача, через програмне переривання **int 21h**.

Функція **Exec** вимагає вказівки ряду параметрів, з яких важливі:

266* Ім'я файлу програми, що запускається. Якщо ім'я не містить шляхи до каталогу, то файл шукається в поточному каталозі, а також у каталогах, перерахованих у перемінної PATH.

267* Командний рядок. Так прийнято називати рядок параметрів, переданих програмі. При запуску програми по команді користувача командний рядок задається після імені програми, вона відділена від імені пробілом. Аналіз умісту командного рядка цілком покладається на програму, що запускається, система лише передає цей рядок програмі.

268* Адреса масиву, що містить параметри середовища програми. Якщо він не заданий, то для програми, що запускається, створюється копія середовища програми-батька.

У MS-DOS використовуються два формати виконуваних програм.

Файл формату COM містить тільки коди позиційно-незалежної програми, що може бути без зміни завантажена для виконання по будь-якій вільній адресі пам'яті. Усі програма повинна міститися в єдиному сегменті, тому розмір файлу обмежений 64 Кб.

Файл формату EXE являє собою переміщувану програму. Файл складається з заголовка,

словника переміщень і власне коду. Інформація в заголовку дозволяє вказати розмір частини файлу, що повинна завантажуватися в пам'ять при запуску програми, максимальний і мінімальний розмір пам'яті, додатково резервируемой для розміщення даних, початкова адреса стека, адреса запуску програми. Розмір файлу практично не обмежений, але розмір частини, що завантажується, повинний бути в межах, наданих DOS, тобто приблизно 500 – 550 Кб.

Перші два байти EXE-файлу містять сигнатуру (ознака) файлу формату EXE, у якості якої використовуються дві букви 'MZ' . Вважається, що це ініціали програміста Марка Збиковского, що брав участь у розробці MS-DOS.

При запуску програми система виконує наступні дії:

269* Виділяє два безперервних блоки пам'яті: для параметрів середовища (блок середовища) і для самої програми (блок PSP). Для програми, як правило, виділяється максимально можливий безперервний блок пам'яті, якщо тільки в заголовку EXE-файлу не заданий менший розмір.

270* Визначає розмір частини програми, що завантажується, (для COM-файлу це весь файл), і зчитує з файлу коди програми.

271* Для EXE-файлу виконує настроювання програми на адресу завантаження, додаючи цю адресу до тих місцям програми, що перераховані в словнику переміщень.

272* Формує на початку блоку програми масив, що називається ***PSP*** (Program Segment Prefix). У PSP утримуються, зокрема, адресу блоку середовища, адреса повернення в батьківську програму, адресу і розмір таблиці JFT, сама ця таблиця, командний рядок програми.

273* Заповнює перші 5 елементів таблиці JFT (стандартні хэндлы), копіюючи них з JFT батьківської програми.

274* Заносить початкові значення в регістри процесора.

275* Запам'ятовує адреса PSP програми як ідентифікатор поточного процесу (PID, Process IDentifier).

276* Нарешті, виконує перехід на адресу запуску програми. Для COM-файлу ця адреса випливає відразу за PSP, для EXE-файлу в заголовку може бути зазначена будь-яка адреса.

4.4.4. Завершення роботи програми

Завершення програми звичайно містить у собі наступні дії системи:

277* Закриваються усі файли, відкриті програмою.

278* Звільняються всі блоки пам'яті, відведені для програми і її середовища.

279* Відновлюються стандартні оброблювачі описаних нижче переривань **`int 23h`** і **`int 24h`**, що могли бути змінені програмою, що відробила.

280* Керування передається батьківській програмі.

У MS-DOS визначені 4 можливі причини, що можуть викликати завершення роботи програми.

281* ***Нормальне завершення***. Воно відбувається, коли програма викликає системну функцію **`Terminate`**. Як аргумент цієї функції програма передає ***код завершення*** – число, що уточнює причину завершення. Звичайно код завершення 0 означає, що програма проробила благополучно, а ненульові значення вказують на різні неприємності (відсутність файлу для обробки, невірний формат даних і т.п.).

282* ***Завершення по запиті користувача***. Воно відбувається, якщо користувач натискає одну з комбінацій клавіш **`Ctrl+C`** або **`Ctrl+Break`**. Система не квапиться припиняти роботу програми, поки не викликає програмне переривання **`int 23h`**. У залежності від результату,

що повертається оброблювачем цього переривання, програма або буде довершена, або продовжить роботу. За замовчуванням система сама обробляє переривання і сама собі відповідає: «Завершити програму». Однак програма може взяти обробку переривання на себе і відповісти системі: «Продовжуємо працювати». Так звичайно і надходять усі серйозні програми MS-DOS.

283* ***Завершення по критичній помилці***. Критичної вважається помилка, що происшли в ході виконання якої-небудь системної функції MS-DOS і имеющая апаратний характер (точніше, виявлена драйвером пристрою). Наприклад, відсутність файлу не є критичною помилкою, а помилка читання з диска – є. Система викликає програмне переривання **`int 24h`**, передаючи як аргументи докладний опис помилки (на якому пристрої, при якій операції і т.п.). У відповідь система хоче одержати від оброблювача вказівка, як реагувати на помилку. Мається 4 види реакції: **`Ignore`** – ігнорувати помилку; **`Retry`** – повторно виконати операцію, що викликала помилку; **`Abort`** – завершити програму; **`Fail`** – повернути код помилки програмі користувача. Системний оброблювач переривання запитує необхідну реакцію в користувача. Однак програма й у даному випадку може взяти обробку переривання на себе і спробувати автоматично вибрати бажану реакцію на підставі опису помилки.

284* ***Завершення з установкою резидента***. Воно схоже на нормальне завершення, але відрізняється двома важливими особливостями. По-перше, файли, відкриті програмою, не закриваються. По-друге, системі повертається не вся пам'ять, займана програмою, що завершилася. Деяка частина пам'яті залишається схованої від системи і може бути використана для розміщення резидентных програм, описаних у наступному пункті.

Мається системна функція, за допомогою якої батьківська програма може перевірити причину і код завершення програми-нащадка.

4.4.5. Перехоплення переривань і резидентные програми

Велика частина усіх функціональних можливостей MS-DOS полягає в обробці різноманітних апаратних і особливо програмних переривань. Зокрема, звертання до численних системних функцій MS-DOS виконується за допомогою виклику програмного переривання **`int 21h`**.

Тим важливіше виявляється той факт, що в MS-DOS програма користувача має можливість перехопити будь-як переривання, тобто установити свій оброблювач цього переривання. Фактично для цього досить записати адреса нового оброблювача по відповідній адресі пам'яті. Маються системні функції для запам'ятовування адреси колишнього оброблювача й установки нового.

Перехоплення апаратних переривань дозволяє програмі оперативно реагувати на різні події. Особливо часто перехоплюються переривання від таймера, що дозволяє виконувати деяку дію регулярно, через заданий інтервал часу, (скажемо, відображати поточний час) або ж виконати його один раз у заздалегідь запланований момент, а також переривання від клавіатури, що дозволяють виконати дія при натисканні визначеної комбінації клавіш. Наприклад, один час були популярні резидентные калькулятори, що з'являлися на екрані при натисканні заданих клавіш. Ще один приклад такого роду – програми, що переключають росіянин/латинський регістри клавіатури.

Перехоплення програмних переривань дозволяє програмі модифікувати виконання будь-якої функції MS-DOS. Вище говорилося про використання перехоплення переривань для визначення реакції програми на натискання **`Ctrl+Break`** і на критичні помилки. Ще одним прикладом може служити системна програма SHARE.EXE, що забезпечує коректний поділ файлів між процесами. Ця програма перехоплює основні файлові функції MS-DOS, щоб

відстежити усі відкриття і закриття файлів і установку/зняття блокувань. На підставі цієї інформації модифіковані функції відкриття, читання і записи файлу визначають, чи дозволена запитана операція.

Програми, що використовують перехоплення переривань, можна розбити на два класи.

285* ***Нерезидентные програми***, що після завершення своєї роботи повертають керування і всю займану пам'ять системі. Такі програми перехоплюють переривання тільки на час своєї роботи і повинні обов'язково відновити стандартну обробку переривань при своєму завершенні. Ця вимога стосується не тільки нормального завершення, але і завершення по **`Ctrl+Break`** і по критичній помилці. У противному випадку при наступному виникненні переривання керування буде передано за адресою вже не існуючого в пам'яті оброблювача, а це крах.

286* ***Резидентные програми*** являють собою оброблювачі переривань, що залишаються в пам'яті і після завершення їхнього процесу, що завантажив, аж до перезавантаження системи. Таким чином, резидентные програми можуть впливати на роботу MS-DOS і всіх програм, що запускаються.

До якого класу повинна належати згадана вище програма SHARE.EXE?

Не буде помилкою сказати, що резидентные програми до деякої міри перетворюють однозадачную MS-DOS у многозадачную систему. У даному випадку реалізується примітивна форма фоново-оперативной диспетчеризації, у якій резидентные програми відіграють роль оперативних процесів, а звичайна робота MS-DOS здійснюється у фоновому режимі.

На жаль, у MS-DOS немає надійною, підтримуваною системою засобів для створення оперативних процесів. Замість цього є тільки можливість перехоплення переривань і ще деякі корисні, але розрізнені функції, що дають прикладному програмістові можливість «вручну» реалізувати многозадачность, але не гарантують коректності результату.

Серед проблем, що приходиться вирішувати розроблювачеві резидентных програм, можна назвати трохи найбільш типових.

287* При установці резидентной програми варто перевірити, чи не була вона уже встановлена, оскільки дворазове перехоплення переривань однієї і тією же програмою звичайно приводить до невірної роботи програми. Таку перевірку можна виконати різними способами. Найчастіше використовують виклик визначеного програмного переривання, що повинне повернути характерний результат, якщо воно вже перехоплено.

288* У MS-DOS немає поняття контексту програми і тим більше немає засобів переключення контексту. Якщо робота резидентной програми може привести до зміни таких системних даних, як поточний диск і каталог, інформація про останній случившейся помилці, ідентифікатор поточного процесу і т.п., то ця програма повинна подбати про збереження і відновлення цих даних, щоб не перешкодити нормальній роботі фонового процесу.

289* Майже усі функції MS-DOS нереентерабельны, причому навіть не по окремості, а в сукупності, тобто виклик однієї з цих функцій при незавершеному виконанні іншої може привести до краху системи. Це не викликає утруднень, поки MS-DOS працює як однозадачная система. Однак активізація резидентной програми може відбутися під час виконання функції DOS, викликаної з фонової програми. Якщо резидентная програма також викликає яку-небудь системну функцію (а без цього неможливо, наприклад, працювати з файлами), то наслідки будуть жалюгідні. Способи обійти це утруднення існують, але по своїй заплутаності вони більше схожі на рецепти алхіміків.

4.5. Керування процесами в Windows

На відміну від «полуторазадачной» MS-DOS, що залишає прикладному програмістові всю роботу (і весь ризик) організації рівнобіжного функціонування процесів, многозадачные ОС надають програмістові більш-менш зручний і багатий набір системних функцій, що дозволяють запустити кілька рівнобіжних процесів і організувати їхня взаємодія (синхронізацію процесів, обмін даними, взаємне виключення і т.п.). При цьому ОС зобов'язана гарантувати коректну й ефективну організацію переключення процесів, поділу між ними процесорного часу, пам'яті й інших ресурсів.

Складність проблеми організації взаємодії рівнобіжних процесів істотно різна для систем, що використовують що витісняє і невитісняє диспетчеризацію процесів. При диспетчеризації, що витісняє, процес може бути перерваний диспетчером практично в будь-який момент. Крім задачі збереження і наступного відновлення контексту процесу (див. п. 4.2.5), яка повинна вирішуватися самої ОС, виникають ще і задачі забезпечення взаємного виключення при виконанні критичних секцій у многозадачных додатках. Тільки розроблювач програми може вирішити, які частини тексту його програми є критичними секціями і повинні бути захищені семафорами.

У системі з диспетчеризацією, що невитісняє, програмістові досить перевірити, що критичні секції не містять викликів що блокують і витісняють функцій. При цьому можна гарантувати, що в ході виконання критичної секції не відбудеться переключення процесів.

Усі версії Windows від 1.0 до 3.11 являли собою досить могутні многозадачные системи з диспетчеризацією, що невитісняє. Версії, починаючи з Windows NT і Windows 95, використовують диспетчеризацію, що витісняє.

4.5.1. Поняття об'єкта в Windows

В ОС Windows широко використовується поняття системного ***об'єкта***. По суті, будь-який об'єкт являє собою деяку структуру даних, розташовану в адресному просторі системи. Оскільки додатка не можуть мати доступу до цієї пам'яті, то для роботи з об'єктом додаток повинний одержати ***хэндл*** об'єкта – деяке умовне число, що буде представляти даний об'єкт при звертанні до API-функцій. Процес одержує хэндл, як правило, при виклику функції **`CreateXxx`** (тут **`Xxx`** – назва об'єкта), що може або створити новий об'єкт, або відкрити існуючий об'єкт, створений іншим процесом. Функції виду **`OpenXxx`** дозволяють тільки відкрити існуючий об'єкт.

Об'єкти Windows поділяються на об'єкти ядра (KERNEL), що дозволяють керувати процесами, об'єкти USER, що описують роботу з вікнами, і об'єкти GDI, що задають графічні ресурси Windows. У даному курсі розглядаються тільки об'єкти ядра. Процеси, нитки і відкриті файли є прикладами об'єктів ядра.

Однієї з відмінних рис об'єктів ядра є ***атрибути захисту***, які можна вказати при створенні об'єкта. Ці атрибути визначають права доступу до об'єкта для різних користувачів і груп. Крім того, при створенні об'єкта ядра можна задати його ім'я, що використовується для того, щоб інші процеси могли відкрити той же об'єкт, знаючи його ім'я.

Хэндл об'єкта може бути використаний тільки тим процесом, що створив або відкрив цей об'єкт. Не можна просто переслати значення хэндла іншому процесові, воно не буде діяти в іншому контексті. Мається, однак, функція **`DuplicateHandle`**, що створює коректну копію хэндла, вимагаючи вказати для цього, який процес створює копію, якого саме хэндла і для якого процесу призначена копія.

Якщо хэндл об'єкта більше не потрібний даному процесові, його варто закрити за допомогою функції **`CloseHandle`**, загальної для різних типів об'єктів.

Об'єкт існує доти, поки не будуть закриті всі хэндлы, що вказують на нього.

4.5.2. Процеси і нитки

Загальне поняття процесу, розглянуте вище в п. 4.2.1, для ОС Windows як би розпадається на два поняття: власне процесу і ***нитки*** (thread; у деяких книгах використовується термін ***потік***). При цьому нитка є одиницею роботи, вона бере участь у конкуренції за процесорний час, змінює свій стан і пріоритет, як було описано вище для процесу. Що ж стосується процесу в Windows, те він може складатися з декількох ниток, що використовують загальну пам'ять, відкриті файли й інші ресурси, що належать процесові. Двома словами: процес – *володіє* (пам'яттю, файлами), нитки – *працюють*, при цьому спільно використовуючи ресурси свого процесу. Правда, нитка теж дечим володіє: вікнами, чергою повідомлень, стеком.

Процес створюється при запуску програми (EXE-файлу). Одночасно створюється одна нитка процесу (повинний же хтось працювати!). Створення процесу виконується за допомогою API-функції **`CreateProcess`**. Основними параметрами при виклику цієї функції є наступні.

290* Ім'я файлу програми, що запускається.

291* Командний рядок, переданий процесові при запуску.

292* Атрибути захисту для створюваних процесу і нитки. І процес, і нитка є об'єктами ядра Windows і в цій якості можуть бути захищені від несанкціонованого доступу (наприклад, від спроб інших процесів втрутитися в роботу даного процесу).

293* Різні прапори, що уточнюють режим створення процесу. Серед них слід зазначити клас пріоритету процесу, прапор отладочного режиму (при цьому система буде повідомляти процес-батько про дії породженого процесу), а також прапор створення припиненого процесу, що не почне працювати, поки не буде викликана функція поновлення роботи.

294* Блок середовища процесу.

295* Поточний каталог процесу.

296* Параметри першого вікна, що буде відкрито при запуску процесу.

297* Адреса блоку інформації, через який функція повертає батьківському процесові чотири числа: ідентифікатор створеного процесу, ідентифікатор нитки, хэндл процесу і хэндл нитки.

Якщо процес успішно створений, функція **`CreateProcess`** повертає ненульове значення.

Клас пріоритету процесу використовується при визначенні пріоритетів його ниток. Докладніше про це в п. 4.5.3.

Хэндл об'єкта ядра Windows (у даному випадку процесу або нитки) дозволяє виконувати різні операції з цим об'єктом. Докладніше про хэндлах і ідентифікатори див. п. 4.5.4.

Після створення процесу його єдина нитка починає виконувати програму процесу, працюючи паралельно з нитками інших запущених процесів. Якщо логіка роботи програми припускає рівнобіжне виконання яких-небудь дій у рамках одного процесу, то можуть бути створені додаткові нитки. Для цього використовується функція **`CreateThread`**. Її основні параметри наступні:

298* атрибути захисту для створюваної нитки;

299* розмір стека нитки;

300* стартова адреса нитки (звичайно нитка зв'язується з виконанням однієї з функцій, описаних у програмі процесу, при цьому як стартову адресу вказується ім'я функції);

301* параметр-покажчик, що дозволяє передати нитки при запуску деяке значення як аргумент;

302* прапор створення нитки в припиненому стані;

303* покажчик на перемінну, у якій функція повинна повернути ідентифікатор створеної нитки.

Значенням функції, що **повертається,** CreateThread є хэндл створеної нитки **або** NULL, якщо створити нитка не удалося.

Прекрасним прикладом многонитевой програми є Microsoft Word. У той час як основна нитка обробляє введення з клавіатури, окрема нитка може динамічно розраховувати розбивку тексту на сторінки, ще одна нитка може в цей же час виконувати печатка документа або його збереження.

Для завершення роботи нитки використовується виклик функції **ExitThread**. Для завершення роботи всього процесу кожна з його ниток може викликати функцію **ExitProcess**. Єдиним параметром кожної з цих функцій є код завершення нитки або процесу.

Завершення процесу приводить до звільнення всіх ресурсів, якими володів процес: пам'яті, відкритих файлів і т.п.

При завершенні процесу завершуються всі його нитки. І навпаки, при завершенні останньої нитки процесу завершується і сам процес.

Не занадто широко відомо, що нитка не є самою дрібною одиницею організації обчислень. Насправді Windows дозволяє створити усередині нитки кілька ***волокон*** (fiber), що у звичайній термінології можуть бути описані як сопрограммы або як задачі з диспетчеризацією, що невитісняє, працюючі в рамках однієї і тієї ж задачі з диспетчеризацією, що витісняє. Переключення волокон виконується тільки явно, за допомогою функції **SwitchToFiber**. Про використання сопрограмм див. /15/.

4.5.3. Планувальник Windows

Задачею планувальника є вибір чергової нитки для виконання. Планувальник викликається в трьох випадках:

304* якщо минає квант часу, виділений поточної нитки;

305* якщо поточна нитка викликала функцію, що блокує, (наприклад, WaitForMultipleObjects **або** ReadFile) і перейшла в стан чекання;

306* якщо нитка з більш високим пріоритетом пробудилася від чекання або була тільки що запущена.

Для вибору нитки використовується алгоритм пріоритетної черги. Для кожного рівня пріоритету система веде чергу активних ниток (тобто ниток, що знаходяться в стані готовності). Для виконання вибирається чергова нитка з непорожньої черги з найвищим пріоритетом.

Черги активних ниток поповнюються за рахунок ниток, що прокинулися після стану чекання і ниток, витиснутих планувальником. Як правило, нитка міститься в кінець черги. Виключення робиться для нитки, витиснутої до витікання її кванта часу більш пріоритетною ниткою. Така «скривджена» нитка ставиться в голову черги.

Значення кванта часу для серверних установок Windows дорівнює звичайно 120 мс, для робочих станцій – 20 мс.

Як ви думаєте, чому для серверів квант часу більше?

Тепер докладніше про пріоритети. Хоча загальна схема їхнього призначення однакова для усіх версій Windows, деталі можуть відрізнятися. Подальший виклад орієнтований на Windows NT 4.0.

Усі рівні пріоритету ниток пронумеровані від 0 (найнижчий пріоритет) до 31 (найвищий). Рівні від 16 до 31 називаються ***пріоритетами реального часу***, вони призначені для виконання критичних за часом системних операцій. Тільки сама система або користувач із правами адміністратора можуть використовувати пріоритети з цієї групи. Рівні від 0 до 15 називаються ***динамічними пріоритетами***.

У Windows використовується двоступінчаста схема призначення пріоритетів. При створенні процесу йому призначається (а згодом може бути змінений самою програмою або користувачем) один з чотирьох ***класів пріоритету***, з кожним з яких зв'язане базове значення пріоритету:

307* ***Realtime*** (базовий пріоритет 24) – вищий клас пріоритету, припустимий тільки для системних процесів, що займають процесор на дуже короткий час;

308* ***High*** (базовий пріоритет 13) – клас высокоприоритетных процесів;

309* ***Normal*** (базовий пріоритет 8) – звичайний клас пріоритету, до якого відноситься велика частина прикладних процесів, що запускаються;

310* ***Idle*** (базовий пріоритет 4) – нижчий (буквально – «неодружений» або «простаивающий») клас пріоритету, характерний для екранних заставок, моніторів продуктивності й інших програм, що не повинні заважати жити більш важливим програмам.

Власне пріоритет зв'язується не з процесом, а з кожною його ниткою. Пріоритет нитки визначається базовим пріоритетом процесу, до якого додається відносний пріоритет нитки – величина від –2 до +2. Відносний пріоритет призначається нитки при її створенні і може при необхідності змінюватися. Мається також можливість призначити нитки критичний пріоритет (31 для процесів реального часу, 15 для інших) або неодружений пріоритет (16 для процесів реального часу, 0 для інших).

Для ниток процесів реального часу пріоритети є статичними в тім змісті, що система не намагається них змінювати за своїм розсудом. Передбачається, що для цієї групи процесів співвідношення пріоритетів повинне бути саме таким, як задумав програміст.

Для процесів трьох нижчих класів їхні пріоритети не випадково називаються динамічними. Планувальник може змінювати пріоритети, а заодно і кванти часу для ниток у ході їхнього виконання, щоб досягти більш справедливого розподілу процесорного часу. Правила зміни динамічних пріоритетів наступні.

311* Коли заблокована нитка дочекалася потрібного їй події, до пріоритету нитки додається величина, що залежить від причини чекання. Це збільшення може досягати 6 одиниць (але пріоритет не повинний перевищити 15), якщо нитка разблокирована внаслідок натискання клавіші або кнопки миші. Таким способом система прагне зменшити час реакції на дії користувача. Усякий раз, коли нитка цілком використовує свій квант часу, збільшення зменшується на 1, поки пріоритет нитки не повернеться до свого заданого значення.

312* Якщо нитка володіє вікном переднього плану (тобто тим, з яким працює користувач), то заради зменшення часу реакції планувальник може збільшити квант часу для цієї нитки з 20 мс до 40 або 60 мс, у залежності від настроювань системи.

313* Якщо планувальник виявляє, що деяка нитка перебуває в черзі більш 3 з, то він підвищує її пріоритет аж до 15 і подвоює її квант. Але ця добродійність разова: коли Золушка-нить витратить збільшений квант або заблокується, її пріоритет і квант повертаються до колишніх значень. Зміст акції зрозумілий: система намагається забезпечити хоч якесь просування навіть для низкоприоритетных ниток.

4.5.4. Процес і нитка як об'єкти

Підсистема керування процесами в Windows дозволяє розглядати процеси і нитки як об'єкти, над якими нитки різних процесів можуть виконати цілий ряд дій. Для цього потрібно мати хэндл процесу або нитки. Як нам відомо, такі хэндлы повертаються як побічні результати роботи функцій **`CreateProcess`** і **`CreateThread`**. Це дозволяє процесові, що запустив інший процес або нитка, виконувати з ними необхідні дії. Однак у деяких випадках бажано дати можливість керувати процесом не процесові-«батькові», а іншому, зовсім сторонньому процесові

(наприклад, програмі адміністрування системи). Проблема при цьому в тім, що, як уже відзначалося, хэндл у Windows має сенс тільки в межах того процесу, що викликав функцію **`Create`** і не може бути просто переданий іншому процесові.

Функція **`OpenProcess`** дозволяє одному процесові одержати хэндл будь-якого іншого процесу, указавши для цього два параметри: ідентифікатор цікавлячого процесу і маску, що задає набір запитуваних прав стосовно цього процесу. Якщо підсистема безпеки Windows, звіривши маркер доступу запитуючого процесу з дескриптором безпеки процесу-об'єкта (див. п. 3.8.4.1), рахує запит законним, то функція поверне необхідний хэндл.

А про яких, власне, правах мова йде? Що саме один процес може зробити з іншим процесом? Основні права наступні:

314* запускати нову нитку процесу;

315* запитувати і змінювати клас пріоритету процесу;

316* читати і записувати дані в пам'яті процесу;

317* використовувати процес як об'єкт синхронізації;

318* примусово припиняти виконання процесу;

319* запитувати код завершення процесу.

Усі перераховані дії виконуються за допомогою відповідних API-функцій (наприклад, для припинення процесу викликається функція **`TerminateProcess`**), одним з параметрів яких указується хэндл відкритого процесу-об'єкта.

4.5.5. Синхронізація ниток

4.5.5.1. Способи синхронізації

Windows надає різноманітні можливості для синхронізації роботи ниток, тобто, інакше кажучи, для організації пасивного чекання ниткою деяких подій, зв'язаних з роботою інших ниток того ж процесу або інших процесів.

320* Традиційної для Windows формою синхронізації є обмін ***повідомленнями*** (messages). При цьому, механізм повідомлень призначений не тільки для синхронізації, але і для обміну даними. Далі робота з повідомленнями буде розглянута докладно.

321* Різноманітні умови чекання можуть бути реалізовані за допомогою ***об'єктів синхронізації*** і ***функцій чекання***, що дозволяють заблокувати нитка до моменту переходу зазначеного об'єкта в сигнальний стан.

322* Використання перемінних типу **`CRITICAL_SECTION`**, на відміну від попередніх способів, можливо тільки для синхронізації ниток того самого процесу, але зате реалізується більш ефективно за часом і пам'яттю.

4.5.5.2. Об'єкти синхронізації і функції чекання

Об'єкти синхронізації являють собою окремий випадок об'єктів ядра Windows. Відмінною рисою об'єктів синхронізації є можливість використовувати як аргументи функцій чекання.

Для створення об'єкта потрібно викликати функцію **`Create`***Xxx*, де замість *Xxx* указується назва конкретного типу об'єктів, наприклад, **`CreateMutex`** або **`CreateEvent`**.

Загальною рисою всіх об'єктів синхронізації є наявність двох станів, одне з яких називається ***сигнальним*** (signaled), а інше, відповідно, несигнальним. Зміст сигнального стану різний для різних типів об'єктів, загальним же є те, що стан чекання для нитки закінчується тоді, коли об'єкт переходить у сигнальний стан.

Для конкретних типів об'єктів замість терміна «сигнальний стан» може бути зручно

використовувати як синоніми «вільно», «включене», «мається», «відбулося» і т.п., а для несигнального, відповідно, «зайнято», «виключено», «ні» і т.п.

Оскільки об'єкти існують у системній пам'яті, процеси можуть одержати до них доступ тільки за допомогою хэндлов. Це вимагає витрат часу на переключення контексту пам'яті при роботі з об'єктами, але зате дає можливість синхронізації ниток, що належать різним процесам. Для цього при створенні об'єктів ядра одним з параметрів функції **Create***Xxx* може вказуватися ім'я цього об'єкта. Якщо після цього інший процес спробує створити об'єкт того ж типу і з тим же ім'ям, то він одержить хэндл вже існуючого об'єкта. У результаті обидва процеси будуть працювати з тим самим об'єктом синхронізації.

До функцій чекання відносяться **WaitForSingleObject** (чекання одного об'єкта) і **WaitForMultipleObjects** (чекання декількох об'єктів). Розглянемо спочатку першу з них.

Функція **WaitForSingleObject** має два аргументи:

323* хэндл якого-небудь об'єкта синхронізації;

324* тайм-аут чекання, тобто максимальний час чекання (у миллисекундах).

Ця функція – що блокує: якщо в момент її виклику зазначений об'єкт знаходиться в несигнальному стані, те нитка, що викликала, переводиться в стан чекання (зрозуміло, пасивного!). Чекання, як правило, завершується в одному з двох випадків: або об'єкт переходить у сигнальний стан, або минає заданий тайм-аут. Третій, особливий випадок завершення зв'язаний з поняттям «закинутий мьютекс», що буде пояснено нижче. Значення, що повертається функцією, дозволяє довідатися, яка з трьох причин пробудження нитки мала місце.

Якщо в момент виклику функції зазначений об'єкт уже знаходився в сигнальному стані, то чекання не відбувається, функція відразу ж повертає результат.

Якщо задано нульову величину тайм-ауту, то виконання функції зводиться до перевірки, чи знаходиться в даний момент об'єкт у сигнальному стані.

Якщо значення тайм-ауту дорівнює константі **INFINITE**, то час чекання не обмежено, тобто нитка може пробудитися тільки по сигнальному стані об'єкта.

Функція **WaitForMultipleObjects** відрізняється тим, що замість єдиного хэндла приймає як аргументи адреса масиву, що містить трохи хэндлов, і кількість цих хэндлов (розмір масиву). Як і в попередньому випадку, задається величина тайм-ауту. Додатковий, четвертий аргумент – булевский прапор режиму чекання. Значення **FALSE** означає чекання *будь-якого* об'єкта (досить, щоб хоч один з них перейшов у сигнальний стан), **TRUE** – чекання *всіх* об'єктів (чекання завершиться, тільки коли всі об'єкти виявляться в сигнальному стані).

Значення, що повертається, дозволяє визначити причину пробудження:

325* перехід одного з заданих об'єктів у сигнальний стан, і якого саме об'єкта;

326* перехід одного з объектов-мьютексов у «закинуте» стан (буде пояснено нижче), і якого саме об'єкта;

327* витікання тайм-ауту чекання.

Крім двох названих, мається ще ряд функцій чекання. Функції **WaitForSingleObjectEx** і **WaitForMultipleObjectsEx** дозволяють також чекати завершення асинхронних операцій уведення/висновку для зазначеного файлу. Функція **MsgWaitForMultipleObjects** дозволяє, поряд з об'єктами синхронізації, використовувати для пробудження поява повідомлень, що очікують обробки. Функція **SignalObjectAndWait** виконує рідкісний трюк: вона переводить один об'єкт у сигнальний стан і відразу починає чекати іншого об'єкта.

4.5.5.3. Типи об'єктів синхронізації

Список об'єктів синхронізації включає наступні типи об'єктів:

328* процеси;

329* нитки;

330* події (у вузькому змісті);

331* мьютексы (двоичные семафори);

332* семафори (недвійкові);

333* таймери;

334* повідомлення про зміни в каталозі;

335* консольне введення.

Розгляд об'єктів синхронізації найпростіше почати з процесів і ниток. Для тих і інших сигнальним станом є только стан завершення роботи. Іншими словами, можна змусити нитка одного процесу очікувати завершення роботи іншої нитки або процесу.

Найпростішим з об'єктів, призначених тільки для синхронізації, є ***подія*** (event). Він створюється за допомогою функції **`CreateEvent`**, що вимагає указати вид події (з автосбросом або з ручним скиданням), а також його початковий стан – сигнальне або несигнальне. Перехід у сигнальний стан виконується функцією **`SetEvent`**, а в несигнальне – **`ResetEvent`**. Функція **`PulseEvent`** як би сполучає обидві попередні: сигнальні стани включається, функція чекання, якщо вона була викликана раніше, завершується, але сигнальний стан відразу скидається.

Подія з ***автосбросом*** означає, що функція чекання при завершенні чекання скидає ту подію (або події), який (яким) вона дочекалася. Якщо кілька ниток чекають того самого події з автосбросом, то разблокирована буде тільки одна з них (Windows не гарантує, яка саме), оскільки іншим «події не дістанеться». Навпроти, подія ***з ручним скиданням*** залишається в сигнальному стані і може бути скинуто тільки функцією **`ResetEvent`**, а доти воно може «нагодувати» скільки завгодно очікують ниток. Можна сказати, що подія з ручним скиданням більше схоже не на *подію*, а на *стан*.

Наступний тип об'єктів – ***мьютекс*** (mutex). Це майже в чистому виді двоичный семафор у змісті Дейкстры. Мьютекс може бути або вільний (це його сигнальний стан), або захоплений якою-небудь ниткою. При створенні мьютекса у функції **`CreateMutex`** указується його початковий стан. Допускається багаторазове захоплення мьютекса *однієї і тією же* ниткою, але після цього нитка повинна стільки ж раз звільнити мьютекс, щоб він повернувся у вільний стан.

Для звільнення мьютекса нитка, що володіє їм, викликає функцію **`ReleaseMutex`**, що є аналогом дейкстровской V(S). Ніяка інша нитка, крім власника, не може звільнити «чужий» мьютекс.

У спеціальній функції для захоплення мьютекса немає необхідності, оскільки роботу примітива P(S) з успіхом виконують функції чекання.

Якщо нитка, що володіє мьютексом, завершує свою роботу, не звільнивши мьютекс, то він переходить у «закинуте» стан: тепер уже ніхто не може його звільнити. Таку ситуацію варто вважати ознакою неакуратного програмування, і вона може бути ознакою наявності в програмі більш серйозних помилок синхронізації.

Об'єкт типу ***семафор*** (semaphore) являє собою недвійковий семафор, використовуваний звичайно для керування розподілом обмеженого числа одиниць ресурсу. У функції **`CreateSemaphore`** указуються два числа: максимальне значення лічильника, зв'язаного із семафором (скільки усього мається одиниць ресурсу), і початкове значення цього лічильника (скільки одиниць вільно). Сигнальним є стан семафора з лічильником більше нуля. Функція чекання зменшує лічильник на 1, а функція **`ReleaseSemaphore`** збільшує його на задану кількість одиниць, але так, щоб результат не перевищував максимального значення.

На відміну від мьютекса, семафор не має власника, тобто можлива і цілком звичайна

ситуація, коли одна нитка захоплює одиниці ресурсу, а інша звільняє них.

Таймер «для чекання» (waitable timer) створюється за допомогою функції **CreateWaitableTimer**, після чого повинний ще бути установлений функцією **SetWaitableTimer**. При створенні таймера вказується його вид: з ручним скиданням або з автосбросом, аналогічно об'єктові «подія». При виклику **SetWaitableTimer** указується час до переходу в сигнальний стан, в одиницях по 100 нс. Час можна вказати або абсолютно (коли «задзвонити»), або щодо поточного моменту (як довго чекати). Можна також указати період (у миллисекундах), через який повинний повторюватися сигнал.

Таймер з ручним скиданням, перейшовши в сигнальний стан, залишається в ньому до виклику функції **CancelWaitableTimer** або повторного виклику **SetWaitableTimer**. Таймер з автосбросом може бути також скинутий функціями чекання.

Корисно нагадати, що в Windows широко використовується й інший тип таймерів «для повідомлень», створюваний функцією **SetTimer**, але ці таймери не можуть взаємодіяти з функціями чекання і навіть не є об'єктами.

Об'єкт типу ***повідомлення про зміни*** створюється за допомогою функції **FindFirstChangeNotification**. При цьому вказуються наступні аргументи:

336* каталог файлової системи, зміни в якому повинні відслідковуватися;

337* прапор, що вказує, чи належні відслідковуватися зміни тільки в заданому каталозі або також у всіх вкладених підкаталогах;

338* фільтр змін, що відслідковуються.

Фільтр змін являє собою маску, у якій окремі біти задають такі події, як зміна списку імен файлів і каталогів (це включає їхнє створення, видалення і перейменування), зміна атрибутів файлу, розміру, дати зміни й атрибутів захисту.

Функція повертає хэндл об'єкта, що переходить у сигнальний стан один раз, з першою появою одного з включених у фільтр змін. Якщо потрібно повторювати відстеження, варто викликати функцію **FindNextChangeNotification**, передаючи їй хэндл. Для видалення об'єкта потрібно викликати функцію з диким ім'ям **FindCloseChangeNotification**.

Повідомлення про зміни дозволяють уникнути активного чекання для програм, що повинні відслідковувати і відображати стан каталогів у динаміку (як це роблять, наприклад, **Провідник** або **Far Manager**). При цьому характер зміни, що происшли, програма може з'ясувати, тільки перечитавши каталог, але зате вірогідно відомо, що якась зміна була.

Хэндл об'єкта типу ***консольне введення*** повертається функцією **CreateFile**, у якої замість імені файлу, що відкривається, зазначений пристрій консольного введення **(CONIN**$). Об'єкт у сигнальному стані, коли в буфері введення є непрочитані символи.

4.5.5.4. Критичні секції

Ще одним засобом синхронізації потоків служать перемінні типу **CRITICAL_SECTION**. По призначенню вони збігаються з розглянутими вище мьютексами, однак реалізація двоичного семафора в цьому випадку зовсім інша. Якщо мьютексы – це один з типів об'єктів ядра, то критичні секції – просто перемінні. У чому тут принципова різниця? У тім, що об'єкти існують у пам'яті системи, тому прикладна програма не може прямо звернутися до об'єкта, прочитати або записати його значення. Програма може тільки одержати хэндл об'єкта, а потім вона доручає системі виконати ту або іншу дію з об'єктом, на який указує даний хэндл. Таке рішення дозволяє використовувати один об'єкт для зв'язку декількох процесів, зберігаючи при цьому ізоляцію пам'яті процесу від пам'яті системи й інших процесів.

Недолік використання об'єктів ядра в тім, що звертання до них усякий раз вимагає

переключення контексту з користувальницького на системний і назад, а це вимагає часу. Така ціна ізоляції процесів.

Якщо потрібно синхронізувати роботу ниток *того самого* процесу, то більш «дешевим» рішенням може бути використання перемінних **`CRITICAL_SECTION`**. Програміст повинний описати перемінну цього типу в програмі процесу, при цьому для всіх ниток процесу буде доступна адреса цієї перемінної. Перед початком роботи з критичною секцією одна з ниток процесу повинна викликати функцію **`InitializeCriticalSection`**, передавши їй адреса перемінної. Потім він може використовувати функцію **`EnterCriticalSection`**, щоб зайняти двоичный семафор, і функцію **`LeaveCriticalSection`**, щоб звільнити його. Як і для мьютекса, той самий потік може кілька разів зайняти критичну секцію, але потім повинний стільки ж раз звільнити неї. Мається ще зручна функція **`TryEnterCriticalSection`**, що дозволяє уникнути блокування і займає критичну секцію, тільки якщо та була вільна в момент звертання. Після закінчення необхідності в синхронізації ниток варто викликати функцію **`DeleteCriticalSection`**.

Виклик усіх перерахованих функцій виконується системою набагато швидше, ніж виклик функцій, що працюють з хэндлами об'єктів, оскільки вся робота виконується в пам'яті зухвалого процесу, без переключення контексту пам'яті. Але зате неможливо використовувати критичні секції для синхронізації ниток *різних* потоків, для цього потрібні мьютексы.

4.5.6. Повідомлення

Величезну роль в організації роботи прикладних програм і самої системи Windows грають ***повідомлення*** (messages). На обробці повідомлень, що посилаються системою, заснована вся робота з графічним інтерфейсом програм. Процеси також можуть обмінюватися повідомленнями з метою синхронізації дій і обміну даними.

Повідомлення являє собою структуру даних, утримуючий тип повідомлення, адресат повідомлення (конкретне вікно, усі вікна або нитка процесу), час посилки повідомлення, координати курсору (для повідомлень від миші), а також два параметри, зміст яких залежить від типу повідомлення.

Система посилає повідомлення додаткам по самих різних приводах: при натисканні клавіш і кнопок миші, створенні і закритті вікон, зміні розмірів і положення вікна, виборі об'єкта в якому-небудь списку, виборі команди в меню і т.п. Часто те саме дія користувача (наприклад, щиглик лівою кнопкою миші на пункті меню) викликає посилку декількох різних повідомлень.

Велика частина повідомлень, що посилаються, може оброблятися самою системою за замовчуванням. Прикладна програма може взяти обробку деяких повідомлень на себе, щоб забезпечити необхідну логікою роботи реакцію на визначені події (наприклад, при зміні розмірів вікна перерозподілити його площа між окремими панелями; при спробі закрити вікно запросити підтвердження і т.п.). Програма повинна також обробляти повідомлення про команди, що посилаються їй, (при використанні меню, кнопок і т.п.) і повідомлення про необхідність перемалювати уміст вікна.

При створенні будь-якого нового вікна Windows вимагає задати ***віконну функцію***, що повинна обробляти повідомлення, спрямовані цьому вікну. Нитку, що одержала повідомлення, або викликає віконну функцію одного зі своїх вікон (для вибору вікна використовується функція **`DispatchMessage`**), або обробляє повідомлення сама (якщо воно не зв'язано з конкретним вікном).

Мається два принципово різних способи посилки повідомлення. Посилаючи повідомлення ***синхронно***, відправник чекає закінчення його обробки, перш ніж продовжити роботу. ***Асинхронна*** посилка нагадує опускання листа в поштову скриньку, вона не робить впливи на

подальшу роботу відправника. Прикладна програма може посилати будь-як повідомлення синхронним або асинхронним способом, як рахує потрібним розроблювач. Для цього можуть використовуватися, відповідно, функція **SendMessage** (синхронна посилка) або функція **PostMessage** (асинхронна посилка).

Нитка процесу одержує повідомлення, адресовані приналежної їй вікнам або самій нитці. Для вибірки повідомлень з черги використовується функція **GetMessage**. У випадку відсутності повідомлень нитка блокується до їхнього надходження. При бажанні програміст може використовувати функцію **PeekMessage**, що перевіряє наявність повідомлення, не блокуючи зухвалу нитку.

Реалізація асинхронного способу посилки досить проста: система просто поміщає повідомлення в чергу нитки-одержувача. Синхронна посилка в ранніх, що невитісняють версіях Windows також реалізувалася дуже просто: як безпосередній виклик віконної функції вікна-одержувача, минаючи всякі черги. Після закінчення обробки керування поверталося відправникові. У сучасних версіях Windows усі так і відбувається, якщо нитка посилає повідомлення своєму власному вікну. В інших випадках таке рішення неможливе. По-перше, якщо повідомлення пересилається між нитками різних процесів, то ізоляція процесів у пам'яті не дає можливості однієї нитки викликати іншу як підпрограму. По-друге, навіть для ниток одного процесу такий виклик міг би привести до непередбачених наслідків у випадку переключення ниток по кванті часу.

У результаті цього системі приходиться діяти дуже акуратно. Нитка, що послала синхронне повідомлення вікну іншої нитки, блокується. Повідомлення міститься в окрему чергу синхронних повідомлень до нитки-одержувача. Ця нитка продовжує свою нормальну роботу до моменту, коли вона звернеться за черговим повідомленням, тобто викликає **GetMessage** або **PeekMessage**. Тоді система викликає віконні функції нитки для обробки кожного із синхронних повідомлень, що очікують. Тільки коли усі вони оброблені, виконується викликана функція **GetMessage** або **PeekMessage**, що перевіряє наявність асинхронних повідомлень у черзі.

У процесі обробки синхронного повідомлення віконна функція не повинна викликати функції, що можуть викликати переключення системи на виконання іншої нитки.

4.6. Керування процесами в UNIX

4.6.1. Життєвий цикл процесу

Поняття процесу в UNIX істотно відрізняється від аналогічного поняття в інших популярних ОС. Якщо в MS-DOS, Windows і інших системах новий процес створюється тільки при запуску програми, то в UNIX створення процесу і запуск програми – це дві зовсім різних дії. При цьому процес у деякому роді «більш первинна», чим програма. Якщо по стандартному визначенню (див. п. 4.2.1) процес є робота з виконання програми, то в UNIX буде більш доречно сказати, що програма – це один з ресурсів, використовуваних процесом.

Існує єдиний спосіб створення процесу в UNIX, і цей спосіб полягає у виклику функції без параметрів **fork()**. Ця функція створює новий процес, що є точною копією процесу-батька: виконує ту ж програму, успадковує такі ж хэндлы відкритих файлів і т.д. При цьому вміст областей пам'яті процесу копіюється. Єдиним розходженням є ідентифікатор процесу (**pid**) – ціле число, унікальне для кожного процесу в системі. Після завершення створення обидва процеси, і батько, і нащадок, будуть виконувати ту саму команду, що випливає в програмі після виклику **fork**. Однак при цьому функція **fork** повертає процесові-батькові значення **pid** породженого нащадка, а нащадкові повертає значення 0. Перевірка повернутого значення – найпростіший спосіб для процесу визначити, «хто він такий» – батько або нащадок.

Типовий фрагмент програми на С може виглядати приблизно так:

```
pid = fork();  // Створення нового процесу
if (pid == -1)   // Процес не створений
   { обробка помилки створення процесу}
else if (pid == 0) // Це породжений процес
   { оператори, виконувані процесом-нащадком }
else               // Це процес-батько
   { оператори, виконувані процесом-батьком }
```

У принципі, обидва процеси можуть надалі виконувати команди з того самого файлу програми. Частіше, однак, процес незабаром після створення починає виконувати іншу програму. Для цього використовується одна з функцій сімейства **exec**. Кілька функцій, імена яких починаються з **exec**, розрізняються деталями – у якому форматі передаються параметри командного рядка, чи варто використовувати пошук файлу програми по перемінної **PATH** і т.п. Кожна з цих функцій запускає зазначений файл програми, однак при цьому не породжується новий процес, що просто текет процес змінює виконувану програму. При цьому цілком перебудовується заново контекст процесу.

Механізм створення нових процесів той самий як для процесів користувача, так і для системних процесів. Єдиним виключенням є самий перший процес, що має ідентифікатор 0. Він породжує другий системний процес, що називається **INIT** і має ідентифікатор 1. Всі інші процеси в системі є нащадками **INIT**.

Для нормального завершення процесу використовується функція **exit(status)**. Ціле число **status** означає код завершення, заданий програмістом, при цьому значення 0 означає успішне завершення, а ненульові значення – різного роду помилки і нестандартних випадків.

Процес-нащадок цілком незалежн від батька, і завершуються ці процеси незалежно друг від друга. Проте, процес-батько має можливість синхронізуватися з моментом завершення нащадка (простіше говорячи, почекати цього завершення). Для цього батько виконує виклик функції **wait**:

```
pid = wait(&status);
```

Ця функція, що блокує, переводить зухвалий процес у чекання до моменту завершення кожного з нащадків, породжених цим процесів. Так працює, наприклад, інтерпретатор команд UNIX, що запускає команду, уведену з консолі, і очікує її завершення. Функція **wait** повертає **pid** нащадка, що завершився, а в **перемінної** status передає код завершення.

Якщо процес-нащадок завершує своє виконання до того, як батько викликав функцію **wait**, то процес, що завершився, переходить у стан, що прийнято називати «зомбі». Фактично від процесу залишається лишь запис у таблиці процесів, що містить код завершення й інформацію про витрачений процесорний час. Усі ресурси, що займалися процесом, звільняються. Якщо надалі батько усе-таки викликає **wait**, то після передачі коду завершення «зомбі» буде викреслений з таблиці, на чому і закінчиться життєвий цикл процесу.

Можлива також ситуація, коли процес-батько завершується до того, як буде довершений його нащадок. Оскільки процес не повинний залишатися без батька, «сирота» буде «усиновлений» системним процесом **INIT**.

Створення нового процесу в системі UNIX є звичайним способом рішення найрізноманітніших проблем. Зокрема, у UNIX відсутні функції асинхронного введення/висновку, вони просто не потрібні. Якщо програміст хоче, щоб операція введення/висновку виконувалася паралельно з роботою основної програми, йому досить

запустити новий процес і доручити йому цю операцію. Замість асинхронного введення/висновку використовується асинхронний паралелізм процесів.

4.6.2. Групи процесів

При вході користувача в систему для нього створюється процес-оболонка, що є предком всіх інших процесів цього користувача. Цей процес стає ***лідером групи*** породжених їм процесів. Як ідентифікатор групи приймається ідентифікатор (**`pid`**) її лідера. Той термінал, з якого користувач ввійшов у систему, стає ***керуючим терміналом*** групи. Це може бути як локальний термінал комп'ютера, на якому працює система, так і вилучений термінал, з якого був виконаний вхід у систему по мережі.

Любою процес може залишити свою групу й оголосити себе лідером нової групи, до якої будуть відноситися його нащадки. Одна з груп є ***поточною*** (foreground), інші групи – ***фоновими*** (background). Процеси поточної групи можуть одержувати введення з керуючого термінала.

Поняття групи процесів відіграє важливу роль у ряді ситуацій при роботі системи. Наприклад, якщо користувач натискає **`Ctrl+C`**, те всім процесам поточної групи посилається сигнал про необхідність завершення. При розриві з'єднання з терміналом подібний сигнал посилається всім процесам, для яких цей термінал був керуючим.

Процес може, створивши власну групу, потім «відкріпитися» від керуючого термінала. Такий процес, називаний у UNIX «***демоном***», утрачає можливість вести діалог з користувачем, але зате він не буде завершуватися, коли користувач закінчить сеанс роботи із системою. Демони в UNIX виконують звичайно загальносистемні задачі, такі, як керування печаткою, одержання і відправлення пошти, автоматичний запуск процесів у заздалегідь задані моменти часу і т.п.

4.6.3. Програмні канали

Одним з «фірмових» винаходів UNIX, згодом запозичених іншими ОС, є поняття ***програмного каналу*** або «трубопроводу» (pipe), що дозволяє виконувати обмін даними між процесами за допомогою тих же системних викликів, що використовуються для читання/запису даних при роботі з файлами і з периферійними пристроями.

Програмні канали можуть бути безіменними або іменованими. Для створення ***безіменного каналу*** процес повинний використовувати системний виклик **`pipe`**, що повертає масив із двох елементів, що містять хэндл для читання з каналу і хэндл для запису в канал. Після цього для роботи з каналом можна використовувати звичайні функції читання з файлу і запису у файл, указуючи відповідні хэндлы каналу. Як правило, процес, що створив канал, потім породжує двох нащадків, з яких один буде виконувати запис у канал, а іншої – читання (нагадаємо, що при створенні процесу він одержує копії всіх хэндлов, відкритих батьком). Не виключена також можливість використання каналу декількома процесами, кожний з яких може, у принципі, як записувати, так і читати дані.

Дані, записувані в канал, буферизуются системою в пам'яті і потім можуть бути прочитані функціями читання з каналу. Якщо в каналі немає даних, то функція читання блокує її процес, що викликав, поки інший процес не запише дані в канал.

Якщо всі процеси закрили хэндлы запису в канал (тобто, немає шансів, що в канал будуть поміщені ще які-небудь дані), те процес-читач, вибравши всі дані, що ще залишалися в каналі, прочитає потім ознаку кінця файлу.

Гірше, якщо закриті всі хэндлы читання, а який-небудь процес намагається виконати запис даних, що комусь буде прочитати. У цьому випадку система посилає процесові сигнал про помилку роботи з каналом.

Використання безіменних каналів має одне обмеження: усі процеси, що працюють з

каналом, повинні бути нащадками процесу, що створив канал. Для передачі даних між неспорідненими процесами можна використовувати ***іменовані канали***, називані також ***каналами FIFO***. Такий канал створюється системним викликом **mknod**, при цьому вказується шлях і ім'я каналу, як при створенні файлу. Імена каналів зберігаються в каталогах файлової системи UNIX нарівні з іменами звичайних і спеціальних файлів. Щоб відкрити канал для читання або для запису, використовується звичайний системний виклик **open** із указівкою необхідного режиму доступу, як при відкритті файлу.

4.6.4. Сигнали

Сигнал являє собою повідомлення процесу про деяку подію, що посилається системою або іншим процесом.

Основними характеристиками сигналу є його номер і адресат сигналу. У різних версіях UNIX використовується від 16 до 32 сигналів. Номер сигналу означає причину його посилки. У програмах номер сигналу звичайно задається однієї зі стандартних констант. Відзначимо наступні сигнали:

339* **SIGKILL** – процес повинний бути негайно припинений;

340* **SIGTERM** – більш «увічлива» форма: система пропонує процесові припинитися, але він може і не послухатися;

341* **SIGILL** – процес виконав неприпустиму команду;

342* **SIGSEGV** – процес звернувся до невірної адреси пам'яті;

343* **SIGHUP** – розірваний зв'язок процесу з керуючим терміналом (наприклад, модем «повісив трубку»);

344* **SIGPIPE** – процес спробував записати дані в канал, до іншого кінцеві якого не приєднаний жоден процес;

345* **SIGSTOP** – процес повинний бути негайно припинений;

346* **SIGCONT** – припинений процес відновляє роботу;

347* **SIGINT** – користувач натиснув **Ctrl+C**, щоб перервати процес;

348* **SIGALRM** – надійшов сигнал від раніше запущеного таймера;

349* **SIGCHLD** – завершився один з нащадків процесу;

350* **SIGPWR** – виникла погроза втрати електроживлення, комп'ютер переключився на автономне джерело (тобто пора терміново рятувати дані);

351* **SIGUSR1**, **SIGUSR2** – номера сигналів, надані в розпорядження прикладних програм для використання в довільних цілях.

При виникненні відповідної події система посилає сигнал процесові або групі процесів (наприклад, **SIGHUP** посилається всім процесам, зв'язаним з даним терміналом).

Любою процес також може послати будь-як сигнал іншому процесові або групі процесів. Для цього використовується системний виклик зі страшним ім'ям **kill**. Параметрами функції служать номер сигналу, що посилається, і одержувач сигналу. Сигнал може бути посланий конкретному процесові (указується **pid** одержувача), а також усім процесам групи відправника або іншої зазначеної групи, або всім процесам, запущеним даним користувачем. Привілейований користувач може навіть послати сигнал усім процесам у системі, за винятком кореневих системних процесів 0 і 1.

Для кожного процесу система зберігає маску, що задає його реакцію на кожний із сигналів. Усього можливо три види реакції.

352* ***За замовчуванням*** – виконуються дії, передбачені системою для даного номера сигналу. Для більшості сигналів дії за замовчуванням передбачають завершення процесу.

353* ***Ігнорувати*** – процес ніяк не реагує на одержання сигналу.

354* ***Обробити*** – у цьому випадку процес повинний задати адресу функції, що буде викликана при одержанні сигналу.

Для сигналів **`SIGKILL`** і **`SIGSTOP`** завжди використовується реакція за замовчуванням. Для інших сигналів процес може установити необхідну реакцію. Традиційним засобом для цього є системний виклик **`signal`**. Одним з параметрів цієї функції вказується номер сигналу, іншим – або адреса функції, що обробляє сигнал, або одне зі спеціальних значень, що означають «За замовчуванням» і «Ігнорувати».

Установка функції-оброблювача для сигналу діє тільки на перший отриманий сигнал з даним номером, відразу ж при виклику цієї функції система автоматично відновлює обробку за замовчуванням. Імовірно, це було зроблено для того, щоб уникнути повторного виклику оброблювача, якщо наступний сигнал буде отриманий до завершення обробки першого. Однак у результаті може случитися інша неприємність: якщо процес не встигне знову установити оброблювач сигналу, то при одержанні наступного сигналу процес може бути припинений у порядку обробки сигналу за замовчуванням.

Ще один недолік традиційної моделі обробки сигналів є неможливість тимчасово заблокувати обробку сигналів на період виконання яких-небудь критично важливих дій. Приходиться вибирати: або сигнал повинний бути оброблений негайно при одержанні, або він буде зігнорований і загублений.

У сучасних версіях UNIX, поряд із традиційними засобами обробки сигналів, може також використовуватися апарат «надійних сигналів», що дозволяє перебороти названі труднощі.

4.6.5. Засобу взаємодії процесів у стандарті POSIX

Десятиліття успішного використання UNIX виявили, проте, ряд недоліків у вихідній архітектурі цієї системи. Одним із самих помітних пробілів була явна слабість механізму синхронізації процесів, заснованого фактично лише на сигналах і на функції **`wait`**. На практиці в більшості реалізацій UNIX уводилися додаткові, більш зручні засоби межпроцессного взаємодії, однак виникала проблема несумісності таких засобів для різних версій UNIX. Різнобій був припинений на початку 90-х років з виробленням стандарту POSIX, що об'єднав усе краще, що на той час було запропоновано в різних версіях UNIX.

До засобів, що, згідно POSIX, повинна підтримувати будь-яка сучасна реалізація UNIX, відносяться, зокрема:

355* сигнали;

356* безіменні й іменовані канали;

357* черги повідомлень;

358* семафори;

359* спільно використовувані (поділювані) області пам'яті.

Сигнали і канали були розглянуті вище. Використання семафорів, черг повідомлень і поділюваної пам'яті дає приблизно ті ж функціональні можливості, що аналогічні засоби Windows, хоча і містить багато цікавих особливостей, що прийдеться, на жаль, залишити за рамками даного курсу.

4.6.6. Планування процесів

4.6.6.1. Стану процесів у UNIX

UNIX є многозадачной системою з пріоритетною диспетчеризацією, що витісняє.

Діаграма основних станів процесу, показана на мал. 4- 1, у випадку UNIX може бути уточнена так, як показано на мал. 4- 2.

Основные состояния процесса в UNIX

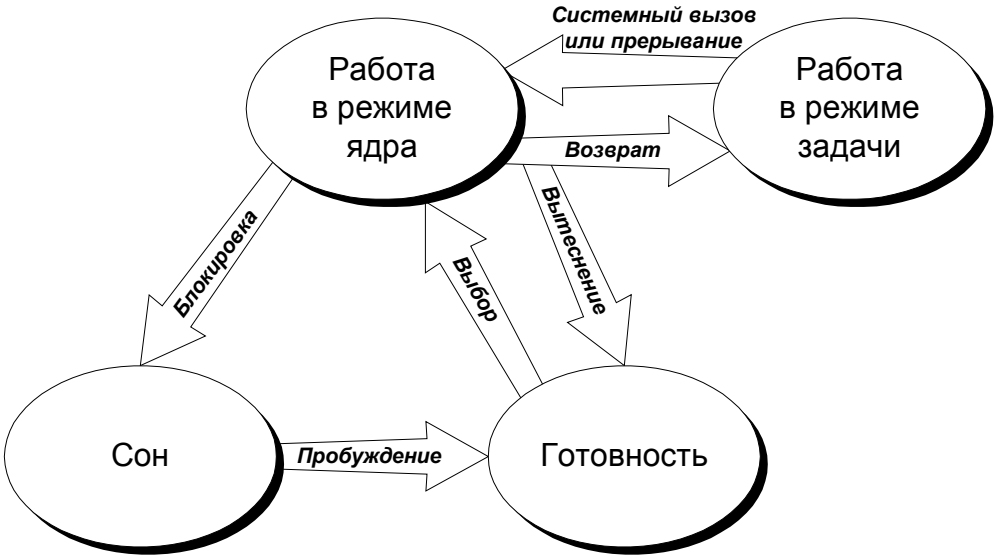


Рис. 4- 2

Поставимо наступне питання: у якому стані знаходиться процес, коли він викликав системну функцію і ядро системи виконує цю функцію? При описі роботи більшості ОС цих питань обходять мовчанням. У UNIX дається чітка відповідь: процес продовжує працювати, але він переходить у режим ядра і виконує системний код. Системні підпрограми, що працюють у режимі ядра, можуть використовувати всі ресурси системи. При цьому контекст процесу залишається доступний, що дозволяє при виконанні системної функції використовувати пам'ять і інші ресурси процесу.

Можливо, що в ході виконання системної функції процес буде заблокований і перейде в стан сну. Після пробудження він перейде в стан готовності. Потім процес буде обраний для виконання і перейде в режим ядра, щоб завершити виконання тієї системної функції, на якій він був заблокований. Після цього процес повинний повернутися в режим задачі, однак відразу після виходу з режиму ядра процес може бути витиснутий, якщо мається активний процес з більш високим пріоритетом.

Ще один шлях зміни стану процесу зв'язаний з обробкою апаратного переривання. При цьому обробка переривань завжди виконується в режимі ядра. Після завершення обробки процесові варто було б повернутися в режим задачі і продовжити виконання прорваної програми. Однак і тут можливі варіанти. Якщо в результаті переривання пробудився більш пріоритетний процес або якщо при обробці переривання від таймера планувальник вибрав інший процес для виконання, то поточний процес буде витиснутий у момент повернення з режиму ядра в режим задачі.

Зі сказаного можна зробити два важливих висновки.

360* Процес, що працює в режимі ядра, не може бути витиснутий ні по витіканню кванта часу, ні при активізації більш пріоритетного процесу. Витиснення процесу можливо тільки в момент повернення з режиму ядра в режим задачі.

361* Перехід у стан сну завжди походить з виконання в режимі ядра. Після пробудження процес повертається в режим ядра через стан готовності.

У діаграмі на мал. 4- 2 не враховані ще деякі стани, у яких може знаходитися процес у UNIX. До їхнього числа відносяться стан старту (процес тільки що створений, але ще не готовий до виконання), стан зомбі (див. п. 4.6.1) і стан припинення, у яке переходить процес, що одержав сигнал **SIGSTOP** (див. п. 4.6.4).

4.6.6.2. Пріоритети процесів

У більшості версій UNIX використовуються рівні пріоритету від 0 до 127. Будемо для визначеності вважати, що 0 відповідає вищому пріоритетові, хоча в деяких версіях справа обстоїть навпаки.

Весь діапазон пріоритетів розділяється на верхню частину (пріоритети режиму ядра) і нижню частину (пріоритети режиму задачі). Цей розподіл показаний на мал. 4- 3.

Уровни приоритетов в UNIX

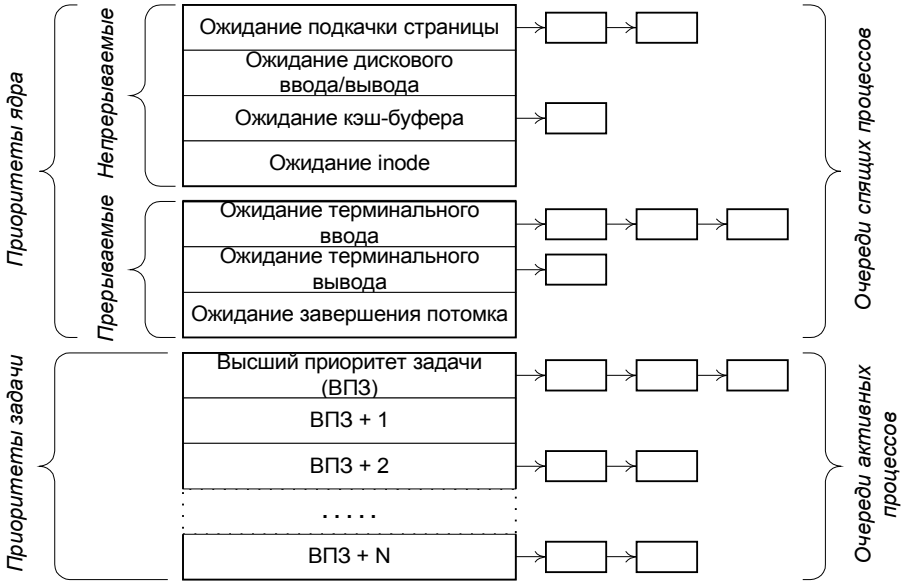


Рис. 4- 3

Поточний (динамічний) пріоритет процесу, що працює в режимі задачі, визначається сумою трьох доданків: базового значення, відносного пріоритету даного процесу і «штрафу» за інтенсивне використання процесорного часу.

Базове значення пріоритету – це те значення, що система за замовчуванням привласнює новому процесові при його створенні. У багатьох версіях UNIX базове значення дорівнює вищому пріоритетові задачі + 20.

Відносний пріоритет, що чомусь називається в UNIX «***nice number***» , привласнюється процесові при його створенні. За замовчуванням система встановлює для процесу нульове значення відносного пріоритету. Звичайний користувач може тільки збільшити це значення (тобто понизити пріоритет процесу), а привілейований користувач може і зменшити аж до вищого пріоритету задачі (для самих пріоритетних процесів). При створенні нового процесу він

успадковує відносний пріоритет батька.

Штраф за використання процесорного часу збільшується для працюючого процесу з кожним перериванням від таймера. Унаслідок цього, высокоприоритетный процес не зможе монополізувати використання процесора і час від часу повинний буде уступати квант часу низкоприоритетным процесам. Однак система «не злопам'ятна»: через щосекунди відбувається зменшення накопичених процесами штрафів наполовину. Таким чином, процес, відлучений від процесора, через якийсь час відновить свій вихідний пріоритет.

Для виконання завжди вибирається активна задача з найвищим пріоритетом, а якщо таких трохи, те вони одержують кванти часу в порядку кругової черги.

Зовсім інший зміст мають пріоритети ядра. Як нам відомо, процеси, що *працюють* у режимі ядра, не можуть бути витиснуті, а тому пріоритети не мають для них ніякого значення. Пріоритети ядра встановлюються тільки для сплячих процесів і залежать тільки від причини сну. У деяких випадках те саме подія приводить до пробудження відразу декількох процесів. У цьому випадку першим починає працювати той, чий пріоритет ядра вище. Розподіл пріоритетів ядра продумано таким чином, щоб першими завершувалися ті системні виклики, що найбільшою мірою блокують використання дефіцитних ресурсів.

Діапазон пріоритетів ядра розділений на двох частин у залежності від того, як реагують сплячі процеси на одержання сигналу. У стані «высокоприоритетного» сну, звичайно зв'язаного з виконанням дискових операцій, процес ігнорує сигнали, що надходять, оскільки їхня обробка могла б затримати реакцію на очікувану важливу подію. Якщо ж процес «спить з низьким пріоритетом», очікуючи події нетермінового і, можливо, нешвидкого (наприклад, натискання клавіші користувачем), то він може прокинутися при одержанні сигналу й обробити цей сигнал.

4.6.7. Інтерпретатор команд shell

Організація інтерфейсу з користувачем звичайно не користується особливою увагою в курсах ОС. Причина цього в тім, що питання інтерфейсу досить далекі від кола проблем, що стосуються інших основних підсистем ОС.

Однак при вивченні UNIX не можна обійти стороною таку цікаву і розвиту частину системи, як інтерпретатор команд, частіше називаний просто ***шелл*** (shell).

Шелл не є частиною ядра UNIX, по своєму статусі це звичайна прикладна програма, що виділяється тільки своїм призначенням, що полягає у виконанні команд користувача, що задаються або в інтерактивному (діалоговому) режимі, або у виді командних файлів, називаних також ***шелл-скриптами***. Існують різні варіанти шелла, що, збігаючись в основному, пропонують трохи різні додаткові можливості.

Набір можливостей, наданих будь-яким інтерпретатором команд UNIX, настільки широкий, що може бути предметом вивчення в окремому курсі. Тут буде дане тільки мінімальне представлення про принципи роботи шелла. Більш підходящою формою вивчення шелла є лабораторні роботи.

Шелл можна розглядати як своєрідна мова програмування, що дозволяє створювати нові програми з досить складними функціями, використовуючи як основні операції виклики інших, більш простих програм. При цьому конструкції мови шелла мають прямий зв'язок з описаними вище засобами керування процесами UNIX.

Базовою конструкцією мови команд UNIX (мови shell) є ***проста команда***. Вона складається з імені команди і, можливо, параметрів, розділених пробілами. Ім'я команди – це звичайне ім'я файлу, що виконується, (або двоичной програми, або шелл-скрипта).

Більшість команд UNIX виводять результат своєї роботи в текстовому виді на стандартний висновок. Як і в MS-DOS, це фактично означає висновок у файл або пристрій, чий хэндл

дорівнює 1. За замовчуванням це означає, що результати виводяться на керуючий термінал користувача. Однак стандартний висновок легко може бути переспрямований у файл або на пристрій. Для цього в команді використовуються символи '**>**' і '**>>**'.

Багато команд використовують також стандартне введення (хэндл 0), що за замовчуванням означає дані, що вводяться з клавіатури термінала. Ознакою кінця введення служить комбінація **Ctrl+D**. Стандартне введення також може бути переспрямований для читання даних з файлу або з пристрою (за допомогою символу '**<**'), або навіть безпосередньо з тексту команди.

Як правило, стандартний висновок команд UNIX має як можна більш регулярну структуру. Наприклад, команда перегляду каталогу **ls -l** видає в кожнім рядку інформацію про один файл, без загального заголовка і без підсумкових даних. Дуже часто висновок команди виглядає як таблиця, стовпці якої розділені знаками табуляції. Це полегшує наступну обробку виведених даних наступними командами. З тих же розумінь команди не видають зайвих повідомлень типу «Команда успішно виконана», хоча можуть видавати повідомлення про помилки.

Для виконання команди шелл запускає окремий процес. У результаті виконання команди виробляється ***код завершення*** процесу, що може потім бути проаналізований. Нульове значення коду звичайно означає нормальне завершення, значення, більше нуля – помилку.

Складена команда складається з простих команд, з'єднаних у виді конвеєра або списку.

Конвеєр означає рівнобіжне виконання декількох команд із передачею даних у міру їхньої обробки від однієї команди до наступної, як на заводському конвеєрі. Запис конвеєра складається з декількох команд, розділених знаками '**|**'. Для виконання конвеєра шелл запускає одночасно працюючі процеси для кожної команди, при цьому стандартний висновок кожної команди перенаправляється на стандартне введення наступної. Фактично для такого перенапрямку використовується механізм програмних каналів, описаний у п. 4.6.3. Перед запуском конвеєра шелл створює необхідна кількість каналів, а при запуску кожного процесу зв'язує його стандартні хэндлы 0 і 1 з відповідними каналами, як показано на мал. 4- 4.

Конвейер в UNIX

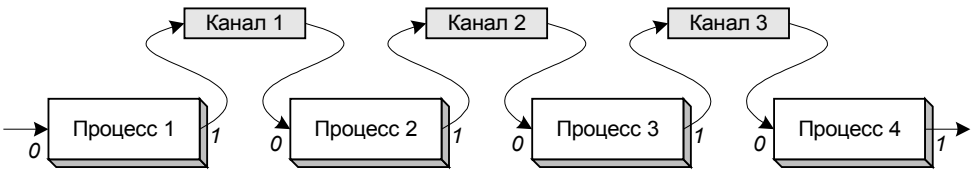


Рис. 4- 4

Список означає послідовне виконання команд. Він складається з декількох команд, розділених знаками '**;**', '**&&**' або '**||**'. Якщо дві команди розділені знайомий '**;**', то наступна команда запускається після завершення попередньої. Якщо команди розділені знайомий '**&&**', то наступна буде виконуватися тільки в тому випадку, якщо код завершення попередньої дорівнює 0 (нормальне завершення). Навпроти, знак '**||**' означає, що наступна команда буде виконуватися тільки в тому випадку, якщо код завершення попередньої не дорівнює 0 (завершення з помилкою).

Якщо запис команди закінчується символом '**&**', то шелл запускає процес її виконання у фоновому режимі, тобто не чекає завершення процесу, а переходить до наступної команди. При цьому фоновий процес продовжує працювати паралельно із шеллом і іншими командами, що запускаються їм. Фоновий процес не має доступу до термінала.

Як при інтерактивній роботі, так і при виконанні скриптов можуть визначатися і

використовуватися перемінні, що мають строкові значення. Ряд перемінних визначається системою, наприклад, **PATH** містить список каталогів, у яких шелл шукає команди, а **HOME** – «домашній» каталог поточного користувача. Для одержання значення перемінної перед її ім'ям записується символ '**$**'. У скриптах можна також використовувати значення параметрів, з якими був викликаний скрипт, від **$1** до **$9**.

Шелл, як і будь-яка мова програмування, містить набір операторів керування порядком виконання команд, таких як **if**, **case**, **while**, **until**, **for**, **break** і деякі інші. Логічні вираження, використовувані в операторах керування, будуються на основі кодів завершення команд, при цьому спеціальна команда **test** дозволяє перевірити різноманітні умови, такі, як існування і тип зазначеного файлу, рівність або нерівність строкових і числових виражень і т.п.

Швидкість виконання шелл-скриптов у багато разів менше, ніж швидкість компилированной програми на C або на іншій мові, однак шелл дозволяє різко спростити рішення багатьох практичних задач, зв'язаних з керуванням операційною системою, обробкою текстових файлів і т.п. Це досягається за рахунок того, що шелл-скрипты дозволяють використовувати «великі блоки»: складати нові програми шляхом витонченого комбінування вже наявних програм-утиліт, набір яких у будь-якій сучасній версії UNIX досить великий.

5. КЕРУВАННЯ ПАМ'ЯТТЮ

5.1. Основні задачі керування пам'яттю

Основна пам'ять (вона ж ОЗУ) є найважливішим ресурсом, ефективне використання якого вирішальним образом впливає на загальну продуктивність системи.

Для однозадачных ОС керування пам'яттю не є серйозною проблемою, оскільки вся пам'ять, не зайнята системою під власні нестатки, може бути віддана в розпорядження єдиного користувальницького процесу. Процедури керування пам'яттю вирішують наступні задачі:

362* виділення пам'яті для процесу користувача при його запуску і звільнення цієї пам'яті при завершенні процесу;

363* забезпечення настроювання програми, що запускається, на виділені адреси пам'яті;

364* керування виділеними областями пам'яті по запитах програми користувача (наприклад, звільнення частини пам'яті перед запуском породженого процесу).

Зовсім інакше обстоять справи в многозадачных ОС. Сумарні вимоги до обсягу пам'яті всіх одночасно працюючих у системі програм, як правило, перевищують наявний у наявності обсяг основної пам'яті. У цих умовах ОС не має іншого виходу, крім почергового витиснення процесів або їхніх частин на диск, щоб використовувати пам'ять, що звільнилася, на нестатки інших процесів. Невдала реалізація такого витиснення може майже цілком застопорити роботу ОС, що велику часть часу буде займатися записом і читанням з диска.

До основних задач, що повинна вирішувати підсистема керування пам'яттю многозадачной ОС, додаються наступні.

365* надання процесам можливостей одержання і звільнення додаткових областей пам'яті в ході роботи;

366* ефективне використання обмеженого обсягу основної пам'яті для задоволення нестатків усіх працюючих процесів, у тому числі з використанням дисків як розширення пам'яті;

367* ізоляція пам'яті процесів, що виключає випадкове або навмисне несанкціоноване звертання одного процесу до областей пам'яті, займаним іншим процесом;

368* надання процесам можливості обміну даними через загальні області пам'яті.

5.2. Віртуальні і фізичні адреси

Поняття «адреса пам'яті» може розглядатися з двох точок зору. З одного боку, при написанні будь-якої програми її автор або явно вказує, по яких адресах повинні розміщатися перемінні і команди (так буває при програмуванні мовою ассемблера), або присвоєння конкретних адрес довіряється системі програмування. Ті адреси пам'яті, що записані в програмі, прийнято називати ***віртуальними адресами***.

З іншого боку, кожній комірці пам'яті комп'ютера відповідає її адреса, що повинна міститися на шину адреси при кожнім звертанні до осередку. Ці адреси називаються ***фізичними***.

В ЕОМ першого покоління не робилося розходження між віртуальними і фізичними адресами: у програмі було потрібно указувати фізичні адреси. Це означало, що така програма могла правильно працювати, тільки якщо сама програма і всі її дані при кожнім запуску (і на будь-якому комп'ютері) повинні були розміщатися по тим самим фізичних адресах. Такий підхід став украй незручним, як тільки була поставлена задача передати розподіл пам'яті під керування ОС.

В даний час програмування у фізичних адресах може використовуватися лише в дуже спеціальних випадках. Як правило, ні програміст, що пише програму, ні компілятор, що транслює її в машинні коди, не повинні розраховувати на використання конкретних фізичних адрес.

Але тоді виникає питання, коли і яким образом повинний відбуватися перехід від віртуальних адрес до фізичного.

Є дві принципово різних відповіді на це питання.

У системах, не розрахованих на використання спеціальних апаратних засобів перетворення адрес, заміна віртуальних адрес на фізичні може бути виконана тільки програмним шляхом. Це повинно бути зроблене до початку роботи програми, або на етапі компонування програми, або (у більш пізніх системах) при завантаженні програми з файлу в пам'ять.

У сучасних системах, призначених для роботи на процесорах із сегментною або сторінковою організацією пам'яті (див. про це нижче), програма навіть після завантаження в пам'ять містить віртуальні адреси. Перетворення у фізичні адреси виконується при вибірці кожної команди з пам'яті, при звертанні до осередків даних – тобто при кожнім використанні адреси. Звичайно, це можливо тільки в тому випадку, якщо мається спеціальна апаратура, що дозволяє перетворювати адреси практично без утрати часу.

5.3. Розподіл пам'яті без використання віртуальних адрес

5.3.1. Настроювання адрес

Якщо в програмі використовуються значення фізичних адрес, то правильність її роботи залежить від того, по яких адресах завантажена в пам'ять сама програма. Це особливо очевидно для команд переходу: якщо в програмі є команда «Перейти за адресою 1000», те зрушення цієї програми в пам'яті приведе до того, що перехід буде виконаний на зовсім іншу команду, хоча і по тій же адресі.

У той же час важко розраховувати, що при кожнім запуску програми ОС зможе завантажити неї по тим самим адресах. Якщо це ще в принципі можливо для однозадачной ОС, то у випадку декількох задач їхньої програми можуть претендувати на ті самі адреси.

Для рішення цієї проблеми в складі файлу програми приходиться зберігати ***словник***

переміщень – список усіх тих місць у програмі, що містять адреси, що вимагають настроювання на адресу завантаження програми. Таке настроювання в більшості випадків являють собою просте додавання адреси завантаження з адресою, що зберігається у файлі програми, і виконується при завантаженні програми в пам'ять. Виконання настроювання приводить до деякої затримки при запуску програм.

Більш пізні архітектури ЕОМ дозволили значною мірою спростити справа за рахунок використання відносної адресації – вказівки адреси як зсуву щодо значення в деякому базовому регістрі. Тепер настроювання була потрібна лише для декількох команд, що завантажують значення в базові регістри. Більш того, для багатьох не занадто складних програм стало можливо обійтися взагалі без словника переміщень (наприклад, якщо всі адреси зазначені тільки як зсуви відносно початку програми). Подібні програми, здатні без змін правильно працювати при завантаженні по будь-якій адресі, називаються ***позиційно-незалежними***, на відміну від ***переміщуваних*** програм, що вимагають настроювання адрес.

У системі MS-DOS усі файли типу COM містять позиційно-незалежні програми, а файли EXE – переміщувані.

5.3.2. Розподіл з фіксованими розділами

При цьому способі розподілу пам'яті адміністратор системи заздалегідь, при установці ОС, виконує розбивку всієї наявної пам'яті на кілька розділів. Як правило, формуються розділи різних розмірів. Допускається також визначення великого розділу як суми декількох примикаючих друг до друга менших розділів, як показано на мал. 5- 1.

Фиксированные разделы

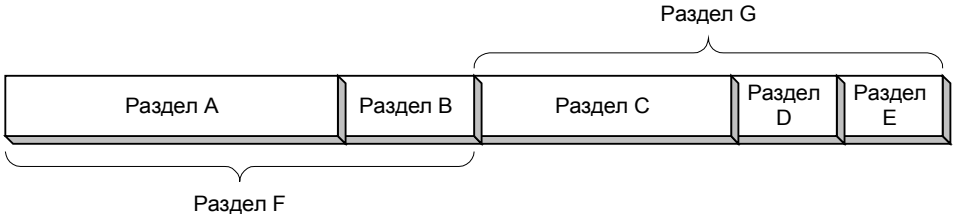


Рис. 5- 1

Можливі два варіанти організації роботи з фіксованими розділами.

369* У більш примітивних системах уже на етапі компонування програми визначається, для якого з розділів вона призначена. У цьому виборі враховується розмір програми, а також, якщо це можливо пророчити, набір інших програм, що будуть працювати одночасно з даної. В ідеалі, паралельно працюючі програми повинні бути скомпоновані для різних розділів. При запуску програми вона завантажується у відповідний розділ, а якщо він зайнятий, програма повинна очікувати його звільнення, тобто повного завершення роботи програми, що займає роздягнув.

370* У більш розвитих системах програма зберігається в переміщуваному форматі, а при її запуску система вибирає найбільш підходящий по розмірі вільний розділ. При відсутності такого програма очікує.

Очевидно, що в обох варіантах пам'ять використовується не занадто ефективно. Цілком можлива ситуація, коли до деяких розділів вишикувалася черга програм, а інші розділи в цей час пустують. У першому варіанті, крім того, неможливий перенос скомпонованих програм на іншу ЕОМ (з іншими розділами).

В описаних варіантах розподілу пам'яті кількість одночасна завантажених програм не може

перевищувати числа розділів. Якщо бажано забезпечити більше задач, то можна дозволити поділ одного роздягнула між декількома програмами. При цьому програма, що у даний момент знаходиться в сплячому або готовому стані, може бути витиснута з пам'яті на диск, у відведений для цього ***файл підкачування*** (swap file). У розділ, що звільнився, з того ж файлу підкачування завантажується програма, що планувальник процесів вибрав для виконання.

Використання підкачування дозволяє необмежено збільшувати кількість завантажених програм (тобто число процесів у системі). Однак підкачування – це досить небезпечне рішення, зловживання нею може в десятки разів знизити продуктивність системи, оскільки велика частина часу буде іти не на корисну роботу, а на перекачування програм з пам'яті на диск і назад. Щоб уникнути цієї небезпеки, у систему повинні бути закладені ретельно підібрані, перевірені алгоритми планування процесів і пам'яті.

Гранично простим випадком розподілу з фіксованими розділами можна вважати організацію пам'яті в однозадачных системах, де існує єдиний розділ для програм користувача, що включає всю пам'ять, не зайняту системою.

5.3.3. Розподіл з динамічними розділами

При такій організації пам'яті ніякої попередньої розбивки не робиться. Уся наявна пам'ять розглядається як єдиний простір, у якому розміщаються завантажені програми. Коли виникає необхідність запустити ще одну програму, система вибирає вільний фрагмент пам'яті достатнього розміру і виділяє його в якості «динамічного розділу» для даної програми. Якщо не вдається знайти досить велика безперервна ділянка пам'яті, то найпростішим рішенням буде почекати з запуском нової програми, поки не завершиться одна з працюючих програм. У принципі, можна використовувати підкачування, але її організація в даному випадку складніше, ніж у випадку фіксованих розділів, оскільки потрібно насамперед вибрати, яка саме з завантажених програм повинна бути витиснута на диск.

Узагалі говорячи, розподіл з динамічними розділами дозволяє більш ефективно використовувати пам'ять, чим при використанні фіксованих розділів. Однак при цьому виникає проблема, що вже зустрічалася нам зовсім в іншій ситуації, у зв'язку з безперервним розміщенням файлів на диску (див. п. 3.3). Мова йде про ***фрагментації***, тобто про розбивку вільної пам'яті на велике число маленьких фрагментів, що не удається використовувати для завантаження великої програми, хоча сумарний обсяг вільної пам'яті залишається досить великим. Фрагментація є неминучим наслідком того, що пам'ять виділяється і звільняється розділами різної довжини, причому в довільному порядку. Але якщо для файлів можна було час від часу виконувати дефрагментацию, переміщаючи усі файли ближче до початку диска, то для працюючих програм це досить важко, оскільки переміщення програми порушило би настроювання адрес, виконану при її завантаженні.

5.4. Сегментна організація пам'яті

Перейдемо тепер до розгляду способів організації пам'яті, що базуються на використанні віртуальних адрес, апаратно преутворених у фізичні при виконанні команд. Два основних види організації віртуальної пам'яті – ***сегментна і сторінкова організація***.

При сегментній організації уся віртуальна пам'ять, використовувана програмою, розбивається на частині, називані ***сегментами***. Ця розбивка виконується або самим програмістом (якщо він програмує мовою ассемблера), або компілятором використовуваної мови програмування. Розміри сегментів можуть бути різними, але в межах максимального розміру, використовуваного в даній архітектурі. Розбивка звичайна виробляється на логічно осмислені частини, такі, як сегмент даних, сегмент коду, сегмент стека і т.п. Велика програма може містити

кілька сегментів одного типу, наприклад, кілька сегментів коду або даних.

Таким чином, при сегментній організації в програми немає єдиного лінійного адресного простору. Віртуальна адреса складається з двох частин: ***селектора сегмента*** і ***зсуву*** від початку сегмента.

Селектор сегмента представляє деяке число, що звичайно є індексом у таблиці сегментів даного процесу. Така таблиця містить для кожного сегмента його розмір, режим доступу (тільки читання або можливе запис), прапор присутності сегмента в пам'яті. Якщо сегмент знаходиться в пам'яті, то в таблиці зберігається його базова адреса (адреса фізичної пам'яті, що відповідає початкові сегмента). Відсутність сегмента означає, що його дані тимчасово витиснуті на диск і зберігаються у ***файлі підкачування*** (swap file).

У кодах програми селектор сегмента може або явно бути присутнім як частина адреси, або матися на увазі в залежності від змісту конкретної адреси. Наприклад, для адрес виконуваних команд використовується селектор поточного сегмента команд, а для адрес операндів – селектор поточного сегмента даних.

При кожнім звертанні до віртуальної адреси апаратними засобами виконується перетворення пари «сегмент : зсув» у фізичну адресу. Спрощена схема такого перетворення показана на мал. 5- 2.

Преобразование сегментного адреса

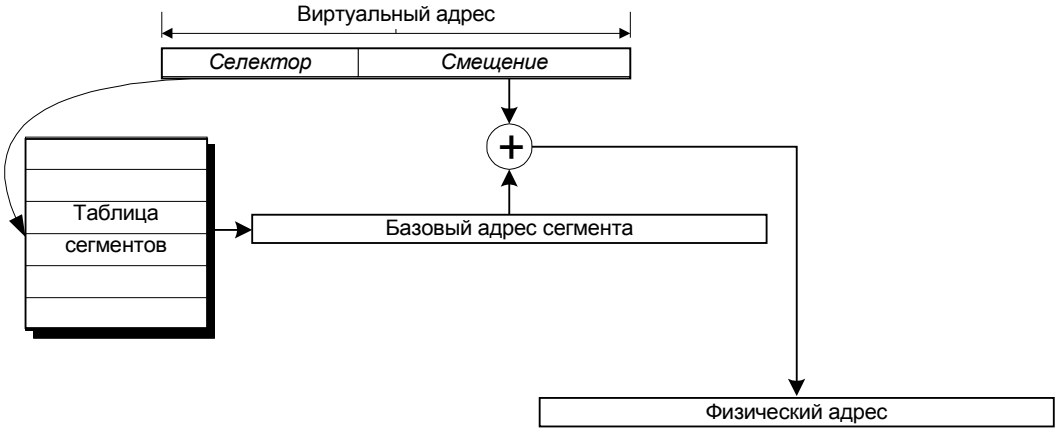


Рис. 5- 2

Селектор сегмента використовується для доступу до відповідного запису таблиці сегментів. Якщо даний сегмент присутній у пам'яті, то його базова адреса, прочитана в таблиці, складається зі зсувом з віртуальної адреси. Результат додавання являє собою фізична адреса, по якому і відбувається звертання до пам'яті.

Якщо сегмент відсутній у пам'яті, то відбувається переривання. Обробляючи його, система повинна довантажити сегмент із диска на вільне місце в пам'яті, записати його базову адресу в таблицю сегментів і потім повторити команду, що викликала переривання.

Але відкіля візьметься вільне місце в пам'яті? Цілком ймовірно, системі прийдеться для цього забрати з пам'яті якийсь інший сегмент, що належить або до цьому ж, або до іншого процесу. Копія сегмента, що витісняється, повинна залишитися у файлі підкачування. Щоб уникнути зайвої роботи, у кожнім записі таблиці зберігається прапор, що відзначає, чи є сегмент у пам'яті «чистим» або «брудним», тобто чи збігається його вміст із дисковою копією або ж воно було змінено в пам'яті після останнього завантаження з диска. «Брудний» сегмент повинний бути збережений на диску, для «чистого» збереження не потрібно. Якщо сегмент визначений як

доступний тільки для читання, то він свідомо «чистий».

Оскільки сегменти мають різні розміри, то в ході роботи системи, що супроводжується багаторазовим завантаженням і вивантаженням сегментів, виникає ефект фрагментації пам'яті, описаний вище в п. 3.3 і в п. 5.3.3. В усіх випадках причиною фрагментації є багаторазове заняття і звільнення областей різного розміру.

Для боротьби з фрагментацією можна час від часу робити дефрагментацию, тобто переміщення всіх сегментів, що знаходяться в пам'яті, на нові місця, без «дірок» у пам'яті між сегментами. При цьому, однак, потрібно, щоб система відкоригувала таблиці сегментів усіх тих процесів, сегменти яких перемістилися у фізичній пам'яті. Крім того, переміщення сегментів забирає відчутний час, тому воно неприпустимо для сегментів, що містять, наприклад, оброблювачі переривань, що повинні спрацьовувати дуже швидко. Щоб уникнути цих проблем, у деяких системах сегменти можуть знаходитися в одному з двох станів:

371* ***фіксований*** сегмент не повинний переміщатися в пам'яті;

372* ***переміщуваний*** сегмент може переміщатися системою, однак програма не може звертатися до адрес у такому сегменті, оскільки його місце розташування не визначено.

Щоб працювати з даними в переміщуваному сегменті, програма повинна попередньо тимчасово зафіксувати його викликом спеціальної системної функції. При цьому ОС визначає поточне місце розташування сегмента і коректує його базова адреса в таблиці сегментів процесу. Якщо ж програма протягом деякого часу не планує звертатися до даного сегмента, то його випливає расфиксировать, оскільки чим більше фіксованих сегментів у системі, тим менш ефективна буде дефрагментация.

При переключенні поточного процесу усе, що повиннао зробити система у відношенні пам'яті, полягає в заміні таблиці сегментів. Для цього або в спеціальний системний регістр записується адреса таблиці сегментів поточного процесу, або, якщо апаратура допускає тільки одну таблицю сегментів, її вміст повинний бути перезаписаний так, щоб відповідати новому працюючому процесові.

5.5. Сторінкова організація пам'яті

Ця форма організації віртуальної пам'яті багато в чому схожа на сегментну. Основні розходження полягають у тім, що всі сторінки, на відміну від сегментів, мають однакові розміри, а розбивка віртуального адресного простору процесу на сторінки виконується системою автоматично. Типовий розмір сторінки – трохи кілобайт. Для процесорів Pentium, наприклад, сторінка дорівнює 4 Кб.

Усі віртуальні адреси одного процесу відносяться до єдиного лінійного простору, Простіше сказати, віртуальна адреса виражається одним числом, від 0 до деякого максимуму. Старші розряди двоичного представлення цієї адреси визначають номер віртуальної сторінки, а молодші розряди – зсув від початку сторінки. Наприклад, для сторінок по 4 Кб зсув займає 12 молодших розрядів адреси.

Фізична пам'ять також вважається розбитої на частини, розміри яких збігаються з розміром віртуальної сторінки. Ці частини називаються ***фізичними сторінками*** або ***сторінковими кадрами*** (page frames). Таблиця сторінок процесу за структурою схожа на таблицю сегментів. Для кожної віртуальної сторінки вона містить режим доступу, прапор присутності сторінки в пам'яті, номер сторінкового кадру, прапор чистоти. Якщо сторінка відсутня в пам'яті, її дані зберігаються у файлі підкачування, що у цьому випадку частіше називають ***сторінковим файлом*** (page file).

Найпростіший варіант схеми перетворення віртуальної сторінкової адреси у фізичну

адресу показаний на мал. 5- 3.

Преобразование страничного адреса

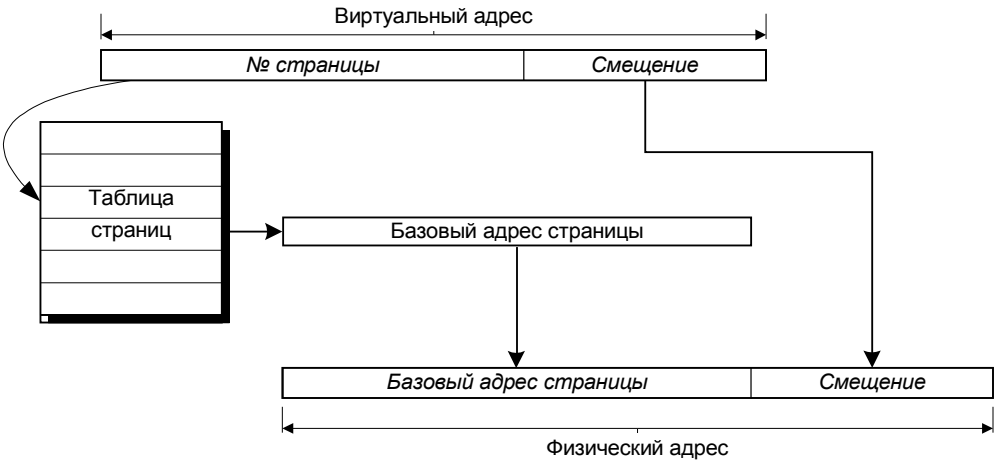


Рис. 5- 3

На відміну від випадку сегментної організації, замість додавання базової адреси зі зсувом у даному випадку можна просто зібрати разом номер фізичної сторінки і зсув.

При переключенні поточного процесу система просто змінює адресу використовуваної таблиці сторінок, тим самим цілком змінюючи відображення віртуальних адрес на фізичні.

Сторінкова організація пам'яті не може привести до фрагментації, оскільки всі сторінки однакові по розмірі, а тому кожна вивільнена фізична сторінка може бути потім використана для будь-якої віртуальної сторінки, що знадобилася.

Розмір простору віртуальних адрес кожного процесу може бути величезним, тому що він визначається тільки розрядністю адреси. Для 32-розрядних процесорів цей розмір дорівнює 2^{32} = 4 Гб. В даний час важко представити програму, який може всерйоз знадобитися стільки пам'яті, та й комп'ютер з таким обсягом пам'яті – річ не рядова . Насправді, програма звичайно використовує лише невелику частину свого адресного простору, не більш декількох десятків або, у крайньому випадку, сотень мегабайт. Тільки ці використовувані сторінки і повинні бути відображені на фізичну пам'ять. Проте, сумарний обсяг сторінок, використовуваних усіма процесами в системі, звичайно перевершує обсяг наявної фізичної пам'яті, тому використання сторінкового файлу стає неминучим.

Керування заміщенням сторінок у фізичній пам'яті в сучасних PC будується за принципом ***завантаження за вимогою*** (demand paging). Це означає наступне. Коли програма тільки лише планує використання визначеної області віртуальної пам'яті (наприклад, для збереження масиву перемінних, описаного в програмі), що відповідають віртуальні сторінки позначаються в таблиці сторінок як існуючі, але, що знаходяться в даний момент на диску. У деяких системах при цьому за віртуальною сторінкою дійсно закріплюються конкретні блоки в сторінковому файлі, хоча з розумінь економії дискової пам'яті це можна зробити пізніше, коли реально буде потрібно записати сторінку на диск. Виділення сторінок фізичної пам'яті не виконується доти, поки програма не звернеться до однієї з осередків віртуальної сторінки. При цьому відбувається апаратне переривання по відсутності сторінки в пам'яті. Це переривання обробляє частина ОС, що називається менеджером пам'яті. Менеджер повинний виконати наступні дії:

373* знайти вільну фізичну сторінку;

374* якщо вільної сторінки немає (а її найчастіше немає), то по визначеному алгоритмі

вибрати зайняту сторінку, що буде витиснута на диск;

375* якщо обрана сторінка «брудна», тобто її вміст змінювався після того, як вона останній раз була прочитана з диска, то «очистити» сторінку, тобто записати неї у відповідний блок сторінкового файлу;

376* на фізичну сторінку, що звільнилася, прочитати блок сторінкового файлу, закріплений за запитаною віртуальною сторінкою;

377* відкоригувати таблицю сторінок, позначивши витиснуту сторінку як відсутню у фізичній пам'яті, а прочитану – як присутню і при цьому «чисту»;

378* повторити звертання до запитаної віртуальної адреси, що тепер уже присутствуют у фізичній пам'яті.

Наступні звертання до віртуальних адрес тієї ж сторінки будуть успішно виконуватися, поки сторінка не буде, у свою чергу, витиснута на диск.

Приведена схема роботи менеджера пам'яті з завантаженням сторінок за вимогою дуже схожа на кэширование диска, розглянуте в п. 2.6.6. Ефективність роботи системи базується на тім же самому ефекті локальності посилань, але тільки застосованому не до блоків диска, а до сторінок пам'яті.

Однак є і дуже істотна відмінність. Звертання програми до дискового кэшу відбувається тільки при запиті на виконання операції читання з диска або запису на диск, що відбувається не настільки часто. Тому система може дозволити собі затратити якийсь час на виконання операцій по підтримці кэша в належному порядку. Наприклад, якщо для вибору блоку, що витісняється, використовується алгоритм LRU, те при кожнім звертанні до кэш-буферу цей буфер повинний переставлятися в кінець черги.

Менеджер пам'яті працює в іншій ситуації. Звертання до пам'яті відбуваються з величезною частотою, при виконанні майже кожної команди процесора. Абсолютно нереально при кожнім звертанні починати якісь програмні дії. З цього випливає, що алгоритм вибору сторінки, що витісняється, повинний спиратися на апаратну підтримку. Оскільки алгоритм LRU не так просто реалізувати апаратно, замість нього часто використовують алгоритми, більш прості в реалізації, нехай навіть вони менш ефективні.

Недоліком сторінкової організації є те, що при великому обсязі віртуального адресного простору сама таблиця сторінок повинна бути дуже великий. При розмірі сторінки 4 Кб і адресному просторі 4 Гб таблиця повинна містити мільйон записів! Однак навряд чи програма процесу постійно використовує весь величезний діапазон адрес. Як правило, на кожнім інтервалі часу інтенсивно використовуються тільки деякі частини таблиці сторінок (це ще один прояв локальності посилань). Бажано мати можливість витісняти на диск тимчасово невикористовувані частини таблиці сторінок. Така можливість у сучасних процесорах забезпечується використанням більш складної, дворівневої схеми сторінкової адресації. У цій схемі весь адресний простір поділяється на розділи рівної величини, кожний з яких описується окремою невеликою таблицею сторінок. Мається також каталог таблиць сторінок, що описує поточний стан кожної таблиці точно так само, як сама таблиця сторінок описує стан сторінок пам'яті. Ті таблиці сторінок, що довго не використовуються, витісняються на диск і відповідним чином позначаються в каталозі. Віртуальна адреса поділяється не на двох, а на три частини. Старші розряди адреси вказують позицію таблиці в каталозі, середні розряди – позицію сторінки в таблиці, молодші – зсув адреси від початку сторінки.

5.6. Порівняння сегментної і сторінкової організації

Обоє розглянутих способу організації віртуальної пам'яті мають свої достоїнства і

недоліки.

До переваг сегментної організації в літературі звичайно відносять наступні.

379* Легко можна вказати режим доступу до сегмента в залежності від змісту його даних. Наприклад, сегмент коду програми звичайно повинний бути доступний тільки для читання, а сегмент даних може бути доступний і для запису.

380* У тому випадку, якщо програма працює з двома або більш структурами даних, кожна з яких може збільшуватися в розмірах незалежно від інших, виділення окремого сегмента для кожної структури дозволяє звільнити програміста від турбот, зв'язаних з розміщенням структур у наявній пам'яті (ці проблеми перекладаються на ОС, що зобов'язана буде знайти місце у фізичній пам'яті для сегментів, що збільшуються,).

381* Набагато рідше називається ще одна, більш прозаїчна причина використання сегментів, що насправді у визначений період була дуже вагомої. Якщо у використовуваній архітектурі комп'ютера розрядність адреси в командах занадто мала (наприклад, 16 розрядів, як у процесорів i286, що дозволяє адресувати усього лише 64 Кб), а розмір програми і її даних досягає багатьох мегабайт, то єдине рішення – використовувати багато сегментів по 64 Кб.

Для сучасних процесорів розрядність адреси складає 32 або навіть 64 біта, що знімає необхідність возитися з великою кількістю дрібних сегментів. При цьому на перший план виходять достоїнства сторінкової організації:

382* програміст не повинний узагалі думати про розбивку програми і її даних на частині обмеженого розміру (сегменти), у його розпорядженні єдиний простір віртуальних адрес;

383* виключається можливість фрагментації фізичної пам'яті і зв'язані з цим проблеми;

384* як правило, зменшується обмін даними з диском, оскільки в нього включаються тільки окремі сторінки, а не цілі сегменти.

Для порівняльної оцінки сегментної і сторінкової організації корисно також згадати історію розвитку версій Windows. Версія Windows 2.0 була орієнтована на процесор i286, що мав сегментну організацію пам'яті з 16-розрядним зсувом у сегменті. В ці роки фірми Intel і Microsoft активно захищали сегментну модель, підкреслюючи її достоїнства. Однак у Windows 3.0 минулого вже частково використані нові можливості процесора i386, а саме, сторінкова організація пам'яті. Оскільки ця версія як і раніше була заснована на 16-розрядних адресах, використання сегментів залишалося необхідним, що привело до складній сегментно-страничної моделі пам'яті. Зате перехід до 32-розрядних версій Windows NT і Windows 95 супроводжувався фактичним відмовленням від використання сегментного механізму на користь чисто сторінкової організації пам'яті. Формально ж тепер весь адресний простір користувача укладається в один дуже великий сегмент розміром 4 Гб.

Великою перевагою використання віртуальної пам'яті, як у сегментному, так і в сторінковому варіанті, є можливість легко і просто ізолювати процеси в пам'яті. Для цього досить, щоб система не відображала ніякі віртуальні сторінки двох різних процесів на ту саму фізичну сторінку. Тоді процеси просто «не будуть бачити» один одного в пам'яті і не зможуть зашкодити один одному.

З іншого боку, у деяких ситуаціях бажано, щоб два або більш процеси мали доступ до загальної області пам'яті. Це дає, наприклад, можливість зберігати в пам'яті єдиний екземпляр системних бібліотек, яким можуть користуватися кілька процесів. Для створення загальної пам'яті досить, щоб віртуальні сторінки всіх зацікавлених процесів відображалися на ті самі сторінки фізичної пам'яті.

5.7. Керування пам'яттю в MS-DOS

MS-DOS – це ОС, що працює в реальному режимі процесора i86, що припускає використання адресного простору розміром усього лише 1 Мб. Насправді, у комп'ютерах IBM гарантується наявність лише 640 Кб основної пам'яті, старшої ж адреси пам'яті зайняті під BIOS

і відеопам'ять, хоча серед них потрапляються розрізнені шматки оперативної пам'яті, називані UMB (верхній блок пам'яті).

Адреса в реальному режимі записується у форматі [*сегмент* : *зсув*], однак тут сегмент – це не селектор, що адресує рядок таблиці сегментів, як описувалося в п. 5.4, а просто номер параграфа пам'яті (1 параграф = 16 байт). Тому можна вважати, що в MS-DOS використовуються тільки фізичні адреси.

У принципі, програми, що працюють у MS-DOS, можуть одержати доступ до пам'яті за межами 1 Мб, але для цього потрібен спеціальний драйвер розширеної пам'яті.

Оскільки поділяти наявну пам'ять між декількома процесами не приходиться, розподіл виходить нехитре. Основні області пам'яті показані на мал. 5- 4.

Распределение памяти в MS-DOS

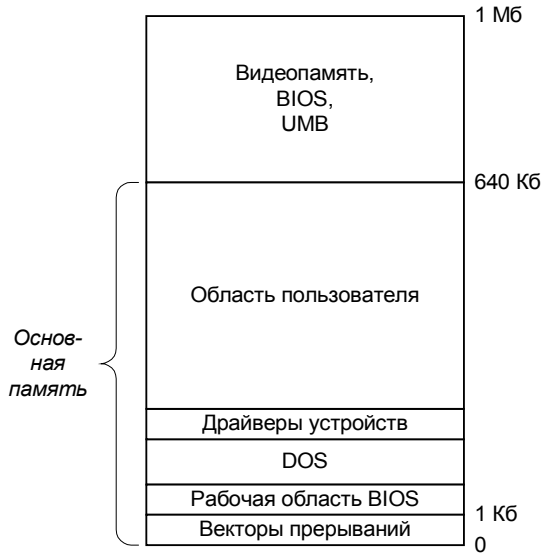


Рис. 5- 4

Нижню частину пам'яті займають модулі ОС: оброблювачі переривань, резидентная чисть інтерпретатора команд, драйвери пристроїв. Деякі системні програми можуть бути заради економії завантажені у верхній блок пам'яті (вище 640 Кб). Усе, що залишається в середині, може бути надано процесові користувача.

Для більшої економії пам'яті деякі нерезидентные модулі DOS можуть займати верхню частину області користувача, але тільки доти, поки не будуть затерті користувальницькою програмою, який буде потрібно вся наявна пам'ять.

Частина системної пам'яті і вся область користувача розбита на прилягаючий друг до друга блоки, розмір яких кратний параграфові. Перед початком кожного блоку пам'яті розміщається блок керування пам'яттю (MCB, Memory Control Block), що займає один параграф і містить наступні дані:

385* ознака, що визначає, чи останній це блок пам'яті або за ним будуть ще блоки (відповідно буква 'Z' або 'M', це, видимо, знову Марко Збиковский відзначився);

386* адреса PSP програми, що володіє цим блоком (0 означає вільний блок);

387* розмір блоку в параграфах;

388* ім'я програми-власника (до 8 символів); це поле надлишкове (знаючи PSP програми, можна знайти ім'я її файлу), воно було додано, імовірно, щоб хоч якось зайняти байти

параграфа, що пустують, MCB.

Коли система повинна виділити блок пам'яті для власних нестатків або по запиті програми користувача, вона переглядає список блоків від початку, переміщаючи від одного MCB до наступного. Знайшовши вільний блок достатнього розміру, система відзначає його як зайнятий відповідним власником. Якщо виділяється не весь вільний блок, то після виділеного блоку система записує ще один MCB, що описує вільний залишок блоку.

При звільненні блоку система записує 0 у поле власника MCB. Якщо з однієї або з двох сторін від блоку, що звільняється, лежать вільні блоки, то два або три вільних блоки зливаються в один.

При запуску програми система виділяє їй два блоки пам'яті: спочатку невеликий блок для перемінні середовища, потім найбільшої серед вільних блоків, що залишилися, для самої програми (блок PSP, див. п. 4.4.3). Звичайно цей блок займає усю вільну пам'ять. Таке рішення прийнятне, оскільки інших претендентів на пам'ять немає.

Чому блок середовища виділяється раніш, ніж блок PSP?

При завершенні програми система переглядає всі блоки пам'яті і звільняє ті з них, власником яких зазначена програма, що завершується. Виключенням є випадок завершення з установкою резидента (п. 4.4.4), при цьому блок PSP не звільняється, але зменшується до зазначеного розміру. Надалі цей блок залишається зайнятим до перезавантаження системи.

MS-DOS надає в розпорядження користувача функції, що дозволяють виконувати основні дії з блоками пам'яті.

389* Виділення блоку зазначеного розміру. Якщо вільного блоку достатньої величини не мається, то система повертає максимальний розмір, що може бути виділений.

390* Звільнення раніше виділеного блоку.

391* Зміна розміру блоку. Зменшення блоку можливо завжди, збільшення – тільки в тому випадку, якщо після даного блоку розташований вільний блок достатнього розміру.

Одним з деяких випадків, коли ці функції виявляються корисні, є запуск породженого процесу. Система повинна мати досить вільного місця, щоб розмістити блок середовища і блок PSP програми, що завантажується. Однак, як було сказано вище, уся вільна пам'ять звичайно віддається під блок PSP поточної програми. Тому перш ніж запускати породжений процес, програма повинна зменшити свій власний блок PSP, залишивши собі необхідний мінімум.

5.8. Керування пам'яттю в Windows

5.8.1. Структура адресного простору

Прийнято вважати, що кожен процес, запущений у Windows, одержує у своє розпорядження віртуальний адресний простір розміром 4 Гб. Це число визначається розрядністю адрес у командах: 2^{32} байт = 4 Гб.

Звичайно, важко розраховувати, що для кожного процесу найдеться така кількість фізичної пам'яті, мова йде тільки про діапазон можливих адрес.

Але навіть і в цьому змісті процесові доступно лише близько 2 Гб молодших адрес віртуальної пам'яті. Зокрема, для Windows NT старші 2 Гб з адресами від 80000000_{16} до $FFFFFFFF_{16}$ доступні тільки системі. Таке рішення дозволило зменшити час, затрачуваний при виклику системних функцій, оскільки відпадає необхідність змінювати при цьому відображення сторінок, потрібно тільки дозволити їхнє використання. Однак, щоб сам виклик API-функцій був можливий, системні бібліотеки, що містять ці функції, розміщаються в молодшій, користувальницькій половині віртуального простору.

У Windows 95 прийняте хуліганське рішення: система і тут розташовується в старшій половині пам'яті, але ця половина доступна процесові користувача і для читання, і для запису. При цьому виклик системи стає ще простіше, але зате система стає беззахисної перед будь-якою некоректною програмою, що лізе куди не треба.

Крім старших 2 Гб, процесові недоступні ще деякі невеликі області на початку і наприкінці віртуального простору. У Windows NT недоступні адреси з 00000000_{16} по $0000FFFF_{16}$ і з $7FFF0000_{16}$ по $7FFFFFFF_{16}$, тобто два шматочки по 64 Кб. Це зроблено з метою виявлення такої типової помилки програмування, як використання неинициализированных покажчиків, що звичайно попадають у заборонні діапазони адрес.

Для 64-розрядних процесорів розмір віртуального адресного простору зростає до важко представимых 2^{64} байт (17 мільярдів гигабайт, якщо завгодно), однак Windows XP виділяє в розпорядження кожного процесу «усього лише» 7152 гигабайта з адресами від 0 до $6FBFFFFFFF_{16}$, а інший адресний простір може використовуватися только системою.

5.8.2. Регіони

Розглянемо тепер, яким образом програма процесу може використовувати свій адресний простір.

Спроба просто-напросто використовувати в програмі довільно обрана адреса в межах адресного простору процесу, швидше за все, приведе до видачі повідомлення про помилку захисту пам'яті. Насправді, використовувати віртуальна адреса можна тільки після того, як йому поставлений у відповідність адреса фізичний. Таке зіставлення виконується шляхом виділення ***регіонів віртуальної пам'яті***.

Регіон пам'яті завжди має розміри, кратні 4 Кб (тобто він містить ціле число сторінок), а його початкова адреса кратна 64 Кб.

Для виділення регіону використовується функція **`VirtualAlloc`**. Вона вимагає вказівки наступних параметрів.

392* Початкова віртуальна адреса регіону. Якщо зазначено константу NULL, то система сама вибирає адресу. Якщо зазначено адресу, не кратний 64 ДО, то система округляє його вниз.

393* Розмір регіону. При необхідності система округляє його до величини, кратної 4 Кб.

394* Тип виділення. Тут вказується одна з констант **`MEM_RESERVE`** (резервування пам'яті) або **`MEM_COMMIT`** (передача фізичної пам'яті), зміст яких буде докладно розглянутий нижче, або комбінація обох констант.

395* Тип доступу. Він визначає, які операції можуть виконуватися зі сторінками виділеної пам'яті. Найбільш важливі наступні типи доступу.

396* **`PAGE_READONLY`** – доступ тільки для читання, спроба запису в пам'ять приводить до помилки.

397* **`PAGE_READWRITE`** – доступ для читання і запису.

398* **`PAGE_GUARD`** – додатковий прапор «охорони сторінок», що повинний комбінуватися з одним з попередніх. При першій же спробі доступу до охоронюваної сторінки генерується переривання, що сповіщає про це систему. При цьому прапор охорони автоматично знімається, так що подальша робота зі сторінкою виконується без проблем.

Найважливіше, що варто зрозуміти про виділення регіонів, це зміст операцій резервування і передачі пам'яті.

Резервування регіону пам'яті (**`MEM_RESERVE`**) означає усього лише те, що діапазон віртуальних адрес, що відповідають даному регіонові, не буде використаний ні під які інші мети,

система вважає його зайнятим. Це як резервування авіаквитка: ви поки що не володієте квитком, але і нікому іншому його не продадуть.

Спроба програми звернутися до адреси в зарезервованому, але не переданому регіоні приведе до помилки.

Передача фізичної пам'яті (**MEM_COMMIT**) означає, що за кожною сторінкою віртуальної пам'яті регіону система закріплює…ні, зовсім не сторінку фізичної пам'яті, як можна подумати. Закріплюється блок розміром 4 Кб у сторінковому файлі. У таблиці сторінок процесу передані сторінки позначаються як відсутні в пам'яті.

Тепер спроба звертання до адреси в регіоні приведе вже до зовсім іншого результату. Оскільки сторінка відсутня в пам'яті, відбудеться переривання. Однак це не буде розглядатися як помилка в програмі. Система, обробляючи переривання, виконає операцію читання сторінки з диска, зі сторінкового файлу, в основну пам'ять і занесе в таблицю сторінок фізична адреса, що тепер відповідає віртуальній сторінці. Після цього команда, що викликала переривання, буде повторена, але тепер уже з успіхом, оскільки необхідна сторінка знаходиться в пам'яті. Подальші звертання до тієї ж віртуальної сторінки будуть виконуватися без проблем, поки сторінка знаходиться в пам'яті.

Резервування і передача пам'яті можуть виконуватися одночасно, при одному звертанні до функції **VirtualAlloc**, що для цього потрібно передати комбінацію обох констант: **MEM_RESERVE** + **MEM_COMMIT**. Є й інший варіант: спочатку зарезервувати регіон пам'яті, а потім, у міру необхідності, передавати фізичну пам'ять або весь регіон відразу, або його окремим частинам (субрегионам). Для цього в перший раз функція **VirtualAlloc** викликається з константою **MEM_RESERVE** і, як правило, без указівки конкретної адреси. Потім викликається **VirtualAlloc** з константою **MEM_COMMIT** і з зазначенням адреси раніше зарезервованого регіону або відповідного субрегиона.

Все описане цілком відповідає поняттю завантаження сторінок за вимогою, описаному в п. 5.5. Як особливості реалізації заміщення сторінок у Windows слід зазначити наступне.

399* Для кожного процесу в системі визначений максимальний і мінімальний розмір його ***робочої безлічі***. При виборі сторінки, що витісняється, система намагається домогтися, щоб за кожним процесом зберігалося не менш мінімального, але не більш максимальної кількості пам'яті. Це дозволяє уникнути ситуації, коли один процес, що марнотратно використовує пам'ять, витісняє з неї майже всі сторінки інших процесів. Процес може змінити розміри своєї робочої безлічі, але при цьому сумарні вимоги всіх процесів обмежуються реальним розміром наявної пам'яті.

400* Процес може ***замкнути в пам'яті*** деякий діапазон адрес, щоб перешкодити витисненню відповідних сторінок на диск. Сумарний розмір пам'яті, замкненої одним процесом, за замовчуванням не повинний перевершувати 30 сторінок. Тривале утримання одним процесом великого числа сторінок замкненими в пам'яті привело б до зменшення обсягу пам'яті, доступного для інших процесів (та й для незамкнених сторінок того ж процесу).

5.8.3. Відображення файлів, що виконуються

При запуску нового процесу Windows використовує практично той же механізм виділення регіонів пам'яті, що був описаний у попередньому пункті. Відмінність у тім, що виділення пам'яті виконує сама система. Крім того, немає необхідності у використанні сторінкового файлу, оскільки для збереження сторінок, витиснутих на диск, як не можна краще підходить сам EXE-файл (або DLL-файл, що містить поділювану бібліотеку функцій). При запуску програми Windows резервує достатня кількість віртуальних сторінок і закріплює за ними блоки EXE-файлу. При цьому немає необхідності виконувати така дія, як «завантаження» програми в

традиційному змісті. Читання кодів програми виконується в міру необхідності, відповідно до загального принципу завантаження сторінок за вимогою.

Як правило, текст програми не піддається змінам у ході її виконання. Це означає, що відповідні віртуальні сторінки залишаються «чистими» і при їхньому витисненні немає необхідності в записі на диск. Якщо той самий файл, що виконується, використовується одночасно декількома процесами (що цілком можливо для EXE-файлу у випадку запуску декількох екземплярів програми, а для DLL-файлів є правилом), то на той самий файл відображаються віртуальні сторінки кожного з цих процесів. При цьому цілком можливо, що в таблицях сторінок різних процесів той самий файл, що виконується, буде відповідати різним діапазонам віртуальних адрес.

Проблема виникає в тому випадку, якщо по логіці роботи програми повинна виконуватися запис у деякі сторінки пам'яті, що відповідають файлові, що виконується. Це нормальне явище, оскільки до складу EXE-файлу можуть входити області пам'яті, відведені для статичних перемінні програми. Очевидно, не можна дозволити програмі змінювати свій власний файл тільки тому, що змінилися значення перемінних. Тим більше, якщо файл використовується відразу в декількох процесах і в кожнім з них перемінні приймають різні значення. Вихід, використовуваний Windows NT у цій ситуації, полягає в наступному. Усі віртуальні сторінки, що відповідають виконуваному файлові, виділяються зі спеціальним атрибутом доступу **`PAGE_WRITECOPY`**, тобто «копіювання при записі». Поки процеси тільки читають дані з цих сторінок, усі відбувається, як описано вище. Якщо ж процес намагається виконати запис на сторінку з таким атрибутом, то відбувається переривання, що система обробляє в такий спосіб. У сторінковому файлі виділяється блок, у який копіюється вміст даної сторінки з EXE-файлу. У таблиці сторінок позначається, що ця віртуальна сторінка тепер закріплена не за EXE-файлом, а за блоком, виділеним зі сторінкового файлу. Після цього операція запису успішно виконується, а при витисненні сторінки вона буде збережена в сторінковому файлі. Таким чином, сторінковий файл усе-таки використовується, але тільки для тих сторінок програми, що змінюються в ході роботи.

Якщо EXE-файл, що запускається, знаходиться на дискеті, то система відразу копіює його в сторінковий файл, оскільки використовувати для підкачування сторінок такий повільний і ненадійний пристрій, як дискетный нагромаджувач, було б украй неефективно.

5.8.4. Файли, відображувані на пам'ять

Неважко зрозуміти, що ті ж засоби, що використовуються для відображення файлу, що виконується, на віртуальну пам'ять, можуть бути застосовані і до будь-якого іншого файлу. У Windows програмістам надається можливість створювати і використовувати об'єкти типу «відображення файлу» (file mapping).

Робота з таким об'єктом вимагає попередньої підготовки. Спочатку програма повинна створити об'єкт, викликавши функцію **`CreateFileMapping`**. Серед параметрів цієї функції можна відзначити:

401* хэндл попередньо відкритого файлу, що буде відображатися на пам'ять;

402* тип доступу до об'єкта (тільки для читання або і для запису);

403* розмір об'єкта;

404* ім'я об'єкта, що може використовуватися для того, щоб різні процеси могли працювати з тим самим об'єктом «відображення файлу».

Функція повертає хэндл створеного або відкритого об'єкта.

Не другому етапі процес викликає функцію **`MapViewOfFile`**, передаючи їй як параметри

хэндл об'єкта «відображення файлу», а також розмір ділянки файлу, що повинний бути відображений, і зсув початку цієї ділянки від початку файлу.

Ця функція повертає віртуальну адресу, що відповідає початкові відображеної ділянки файлу в пам'яті процесу. Іншими словами, виявляється, що задана ділянка файлу якимсь образом вже очутився в пам'яті, хоча функція читання з файлу не викликалася.

Насправді, звичайно, дані з файлу ще не знаходяться в пам'яті. Просто в таблиці сторінок відзначено, що за так-те сторінками віртуальної пам'яті закріплені блоки файлу. Цього разу не сторінкового файлу і не EXE-файлу програми, а того файлу, що використовувався при створенні об'єкта «відображення файлу». Далі починає діяти усі той же стандартний механізм завантаження сторінок за вимогою, тобто реальне читання з файлу відбудеться тоді, коли програма звернеться до віртуальних адрес, що відповідають відображеній ділянці файлу.

Зі сказаного випливає, що відображення файлу на пам'ять являє собою оригінальний спосіб роботи з файлами, при якому замість явного читання і запису даних програма поводиться так, ніби файл уже знаходився в пам'яті. Це дійсно так, але це ще не усі.

Якщо два процеси працюють з тим самим об'єктом «відображення файлу» і ділянки файлу, що вони відображають у свою пам'ять, хоча б частково перетинаються, то Windows гарантує, що будь-які зміни даних, зроблені на цій ділянці одним процесом, зараз же стають доступними іншому процесові. Таким чином, два процеси фактично працюють із загальною областю пам'яті, як показано на мал. 5- 5.

Отображение файла в память процессов

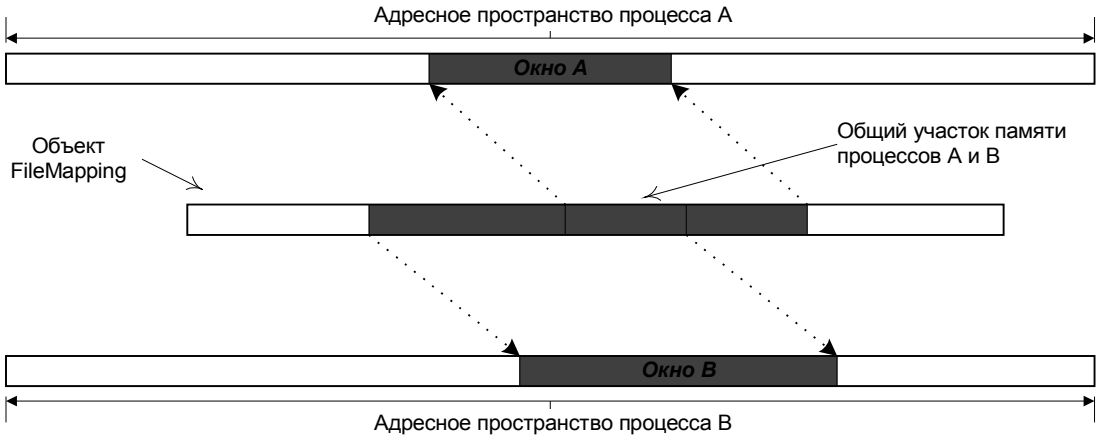


Рис. 5- 5

З цього погляду, відображення файлів можна розглядати як один із засобів взаємодії процесів, що дозволяє під контролем системи перебороти ізоляцію пам'яті процесів.

Саме забавне, що файл-те тут і не обов'язково. Якщо при створенні об'єкта «відображення файлу» у якості хэндла файлу зазначене спеціальне значення $FFFFFFFF_{16}$, то Windows зв'язує сторінки пам'яті з блоками сторінкового файлу. У цьому випадку об'єкт може використовуватися тільки як засіб обміну даними між процесами, без прив'язки до конкретного файлу. При закритті об'єкта «відображення файлу» його дані в цьому випадку не зберігаються.

5.8.5. Стеки і купи

Описані вище засобу керування пам'яттю, засновані на виділенні регіонів, являють собою могутній і красивий інструмент для роботи з великими масивами пам'яті. Однак у практиці програмування частіше зустрічаються прозаїчні задачі, зв'язані з використанням невеликих

ділянок пам'яті: виклик функцій з передачею їм параметрів і виділенням локальних перемінних, створення і звільнення перемінних у динамічній пам'яті і т.п. Але зате ці дрібні операції можуть виконуватися дуже багато разів. Використовувати виділення окремого регіону заради того, щоб одержати 10 – 20 байт пам'яті, це приблизно ту ж саме, що застосовувати ракетну зброю в боротьбі з тарганами. Нагадаємо, що мінімальний розмір регіону дорівнює 4 Кб.

Програмісти звикли, що для розміщення параметрів і локальних перемінних функцій використовується стек, а виділення ділянок динамічної пам'яті походить з «купи» – спеціально призначеної для цього області пам'яті, керованою виконуючою системою мови програмування. Обоє цих механізму одержують підтримку з боку Windows.

При створенні нової нитки вона одержує свій власний стек, розмір якого, якщо він не зазначений явно, приймається рівним 1 Мб. Ця величина може показатися явно надлишкової для більшості програм. Чи коштує системі так шпурлятися пам'яттю?

Насправді, виділяється 1 Мб *резервованої* пам'яті. Реальної витрати ресурсів пам'яті при цьому не відбувається, хіба що від 2 Гб адресні простори процесу отщипывается порівняно невеликий шматочок. Розмір «дійсної» пам'яті, закріпленої за стеком у сторінковому файлі, дорівнює спочатку двом сторінкам, розміщеним у самому кінці зарезервованого регіону, як показано на мал. 5- 6.

Выделение памяти для стека

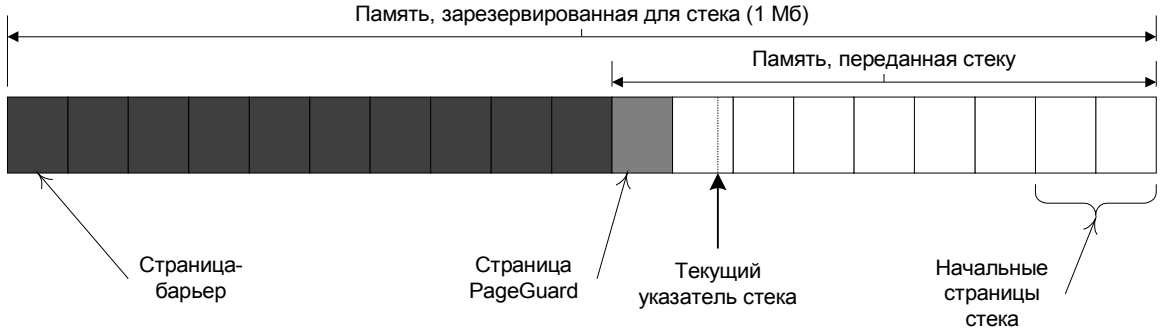


Рис. 5- 6

Стік по своєму звичаї росте в напрямку убування адрес, тому, починаючи ріст із самої старшої адреси, він поступово заповнює одну сторінку, а потім переходить до другого, з меншими адресами. Ця сторінка виділяється з атрибутом **`PAGE_GUARD`**, що, нагадаємо, означає, що при першому звертанні до сторінки генерується переривання, що оповіщає про це систему. Для Windows це сигнал, що настав час передати стеку ще одну сторінку пам'яті про запас, причому ця сторінка знову одержує атрибут **`PAGE_GUARD`**, щоб потім сповістити систему, що стік знову виріс. Так може продовжуватися доти, поки пам'ять не буде передана всім сторінкам регіону стека, крім самої молодшої. Ця сторінка завжди залишається зарезервованої і спроба запису на неї розглядається як переповнення стека. Таким чином, молодша сторінка регіону стека служить бар'єром, що не дозволяє стеку вийти за початок регіону.

А чому необхідний такий бар'єр?

Для розміщення динамічних перемінних зручно використовувати об'єкт «купа» (heap). Windows надає кожному процесові власну купу, хэндл якої можна одержати викликом функції **`GetProcessHeap`**. Після цього нитки процесу можуть запитувати блоки пам'яті з купи, викликаючи функцію **`HeapAlloc`**. Параметри цієї функції містять у собі хэндл купи, розмір запитуваного блоку і деякі прапори. Після закінчення потреби у виділеному блоці він може бути

повернутий у купу викликом функції **HeapFree**.

Невелика проблема виникає в зв'язку з тим, що до одній і тій же купі можуть звертатися різні нитки одного процесу. Не виключено їхнє одночасне звертання до функцій, що працюють з купою. Виникає проблема взаємного виключення, і Windows вирішує неї, використовуючи убудований мьютекс. Це називається ***сериализацией*** доступу до купи, тобто послідовним виконанням запитів (від «serial» – послідовний). Для програми користувача цей мьютекс не видний і можна не звертати на нього уваги. Однак у тому випадку, якщо програміст упевнений, що нитки не можуть перешкодити один одному, він може відключити сериализацию, указавши відповідний прапор або при відкритті купи, або при запиті блоку. Це трохи підвищує продуктивність.

У деяких випадках виявляється вигідно використовувати не одну, а кілька куп для того самого процесу. Можна назвати, принаймні, дві подібних ситуації.

405* Якщо з купою працюють дві або більш нитки процесу, то виділення окремої купи для кожної нитки дозволяє обійтися без сериализации, підвищивши в такий спосіб продуктивність.

406* Якщо програма запитує з купи блоки різного розміру, то неминуче таке знайоме нам явище, як фрагментація пам'яті. У даному випадку вона приведе до зайвого росту купи і до уповільнення роботи. Іноді вдається уникнути фрагментації, виділивши окрему купу для кожного використовуваного розміру блоків. Наприклад, з однієї купи будуть запитуватися блоки тільки розміром 105 байт, а з іншої – розміром 72 байта. При виділенні блоків одного розміру фрагментації не виникає.

Windows дозволяє процесові створити будь-як кількість додаткових куп. Для цього потрібно викликати функцію **HeapCreate**, передавши їй два числа: початковий розмір фізичної пам'яті, переданій купі при створенні, і максимальний розмір купи, що задає розмір регіону зарезервованої пам'яті для купи. Якщо максимальний розмір заданий рівним 0, то купа може рости необмежено.

Коли додаткова купа перестає бути потрібна, можна звільнити займану пам'ять, передавши хэндл купи функції **HeapDestroy**.

5.9. Керування пам'яттю в UNIX

Дати загальну характеристику керування пам'яттю в UNIX важко, оскільки ця частина системи перетерпіла найбільші зміни за довгий період існування UNIX, пройшовши шлях від керування динамічними розділами фізичної пам'яті до сучасної схеми заміщення сторінок за вимогою, подібної тієї, котра використовується в Windows.

Кожен процес у UNIX має власний віртуальний адресний простір, що розділяється на області програми, даних і стека. У залежності від архітектури пам'яті комп'ютера, область пам'яті UNIX може бути реалізована як один або кілька сегментів пам'яті, як набір віртуальних сторінок або просто як область у фізичній пам'яті. Адреси і розміри областей зберігаються як частина контексту процесу.

Область програми, як правило, доступна тільки для читання і має фіксований розмір. Область даних доступна для читання і запису, її розмір може змінюватися в ході роботи за допомогою системного виклику **brk**. Область стека автоматично збільшується системою в міру заповнення стека. При запуску декількох процесів, що виконують ту саму програму, вони розділяють загальну область програми, але кожний із процесів одержує власні області даних і стека.

Коли процес виконує системний виклик **exec** (тобто починає виконувати іншу програму),

усієї його області пам'яті звільняються і потім виділяються заново.

Ядро системи має власні області пам'яті. При виконанні системних викликів ядро працює в контексті процесу, що викликав, тобто воно має доступ як до власної пам'яті, так і до областей пам'яті процесу.

Ефективне використання обмеженого обсягу фізичної пам'яті забезпечується роботою системних процесів-«демонів», що керують переміщенням інформації між пам'яттю і файлом підкачування. У ранніх реалізаціях UNIX, що не використовували механізм віртуальних сторінок, була можлива тільки підкачування і витиснення цілих процесів. Демон підкачування (системний процес з ідентифікатором 0) періодично відслідковував стан процесів і приймав рішення про витиснення окремих процесів на диск і підкачуванню в пам'ять, що звільнилася, одного з раніше витиснутих процесів. При цьому в увагу приймалася тривалість перебування процесу в пам'яті або на диску, що текет стан процесу (сплячі процеси – більш підходящі кандидати на витиснення, чим готові) і його розмір (часто тягати сюди великі процеси невигідно).

У сучасних реалізаціях, заснованих на сторінковій організації пам'яті, значну роль грає поняття списку вільних сторінок, що можуть бути негайно виділені, якщо який-небудь процес звернеться до віртуальної сторінки, відсутньої в пам'яті. У цей список заносяться сторінки, до яких довго не було звертання з боку процесів. Оскільки у відношенні сторінок пам'яті важко реалізувати алгоритм LRU, звичайно використовується більш простий алгоритм «другого шансу» (його інша назва – «алгоритм годин»). Ідея полягає в наступному. Для кожної фізичної сторінки в таблиці сторінок зберігаються біт використання і біт модифікації. Ці біти встановлюються апаратно: біт використання – при кожнім звертанні до сторінки, а біт модифікації – при записі на сторінку. Системний процес, називаний демоном заміщення сторінок, активізується періодично (по таймері) і стежить, чи не занадто малий розмір списку вільних сторінок. Звичайно поріг установлюється рівним ? загального обсягу фізичної пам'яті. Якщо число вільних сторінок нижче цього порога, демон починає циклічно перевіряти усі фізичні сторінки. Якщо в сторінки встановлений біт використання, то цей біт скидається. Якщо ж біт уже був скинутий, то сторінка включається в список вільних. Таким чином, сторінка попадає в список вільних, якщо після останнього звертання до неї сторінковий демон устиг двічі неї опитати. Якщо в сторінки встановлений біт модифікації (в іншій термінології, якщо сторінка «брудна»), то перед її зарахуванням у список вільних система зберігає дані в сторінковому файлі. Зарахування в список вільних сторінок не означає негайної втрати даних, і якщо процес устигне звернутися до сторінки до того, як вона буде віддана іншому процесові, ця сторінка буде виключена з числа вільних.

Демон підкачування процесів теж не дрімає. Якщо сторінковий демон занадто часто виконує запис сторінок на диск, а число вільних сторінок проте залишається низьким, то демон підкачування вибирає один із процесів і відправляє його цілком на диск, щоб зменшити конкуренцію за пам'ять.

Тут описаний (досить приблизно) лише один з варіантів керування пам'яттю, реалізованих у різних версіях UNIX і в Linux. Оскільки алгоритми керування фізичною пам'яттю є «внутрішньою справою» системи і не регламентуються ніякими стандартами, що відповідають алгоритми, використовувані в різних версіях системи, можуть значно відрізнятися.

6. ЛІТЕРАТУРА

1. Олифер В.Г., Олифер Н.А. Мережні операційні системи. Спб.: Питер, 2001. 544 с.
2. Таненбаум Э. Сучасні операційні системи. Спб.: Питер, 2002. 1040 с.

3. Столлингс В. Операційні системи. М.: Вільямс, 2002. 848 с.
4. Бек Л. Введення в системне програмування. М.: Світ, 1988. 448 с.
5. Краков'як С. Основи організації і функціонування ОС ЕОМ. М.: Світ, 1988. 480 с.
6. Кейслер С. Проектування операційних систем для малих ЕОМ. М.: Світ, 1986. 680 с.
7. Шоу А. Логічне проектування операційних систем. М.: Світ, 1981. 360 с.
8. Ріхтер Дж. Windows для професіоналів. М.: Видавничий відділ «Російська редакція» ТОО “Channel Trading Ltd.”, 1995. 720 с.
9. Питрек М. Секрети системного програмування в Windows 95. Київ, Діалектика, 1996. 448 с.
10. Питрек М. Внутрішній світ Windows. Київ, «Диасофт Лтд.», 1995. 416 с.
11. Робачевский А.М. Операційна система UNIX. Спб.: БХВ, 1999. 528 с.
12. Дансмур М., Дейвис Г. Операційна система UNIX і програмування мовою Си. М.: Радіо і зв'язок, 1989. 192 с.
13. Бах М. Архітектура операційної системи UNIX.
14. Финогенов К.Г. Самовчитель по системних функціях MS-DOS. М.: Радіо і зв'язок, Энтроп, 1995. 382 с.
15. Батіг Д. Мистецтво програмування. Т.1, Основні алгоритми. М.: Вільямс, 2002. 720 с.
16. http://www.citforum.ru
17. http://www.rsdn.ru
18. http://www.emanual.ru
19. http://www.infocity.kiev.ua
20. http://www.helloworld.ru
21. http://www.msdn.microsoft.com
22. http://www.sysinternals.com
23. http://www.bcd.org
24. http://www.linux.org
25. http://www.informit.com