

О.М. Хошаба

***ОПЕРАЦІЙНІ СИСТЕМИ
КОМП'ЮТЕРНИХ МЕРЕЖ***

Міністерство освіти і науки України
Вінницький національний технічний університет

О.М. Хошаба

ОПЕРАЦІЙНІ СИСТЕМИ КОМП'ЮТЕРНИХ МЕРЕЖ

Затверджено Вченою радою Вінницького національного технічного університету як навчальний посібник для студентів напряму підготовки 0804 - "Комп'ютерні науки" спеціальностей 7.080403 - "Програмне забезпечення автоматизованих систем" та 7.080004 - "Інтелектуальні системи прийняття рішень". Протокол № 3 від 30 жовтня 2003 р.

Рецензенти:

В.П.Кожем'яко, доктор технічних наук, професор

С.М.Захарченко, кандидат технічних наук, доцент

І.І. Хаїмзон, доктор технічних наук, професор

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України

О.М. Хошаба

X 87 **Операційні системи комп'ютерних мереж.** - Вінниця: ВНТУ, 2004.- 115 с.

В навчальному посібнику значна увага приділяється вивченню мережевих механізмів операційних систем з використанням комунікаційних методів. Подані приклади роботи програм з мережевими структурами операційних систем на мовах високого рівня C та Java. Описані методи роботи з функціями введення/виведення мережевої операційної системи Unix. Розглянута загальна характеристика відомих мережевих операційних систем FreeBSD та Solaris. Посібник розроблений у відповідності з планом кафедри та програмами дисциплін "Операційні системи комп'ютерних мереж".

УДК 621.3.06(075)

Зміст

Вступ	6
Розділ 1. Загальні відомості про операційну систему Unix	9
1.1. Ядро і процеси	10
1.2. Диспетчер мережевої операційної системи UNIX	12
1.3. Процеси мережевої ОС UNIX	13
1.3.1. Дескриптор і контекст процесу	13
1.3.2. Ієрархія процесів	13
1.3.3. Ідентифікатори процесу	15
1.3.4. Групи процесів ОС UNIX	15
1.3.5. Взаємодія процесів ("батьки", "діти", "сироти", "зомбі")	16
1.3.6. "Процеси-демони"	17
Контрольні питання	18
Контрольні вправи	18
1.4. Міжпроцесні комунікації	19
1.4.1. Сигнали	19
1.4.2. Семафори	22
1.4.3. Програмні канали	23
1.4.4. Черги повідомлень	30
1.4.5. Розподілена пам'ять	31
Контрольні питання	32
Контрольні вправи	32
1.5. Багатокористувачевий захист	33
1.5.1. Ідентифікатори користувача і групи	33
1.5.2. Привілейований користувач	34
1.5.3. Ефективні і реальні ідентифікатори	34
1.6. Знайомство з механізмом свопінгу та пейджингу	35
1.6.1 Загальний механізм роботи	35
1.6.2 Буфер кеш-пам'яті	38
1.6.3 Переваги і недоліки буферного кешу	39
Контрольні питання	40
Контрольні вправи	41
1.7. Керування введенням/виведенням	41
1.7.1. Спеціальні файли операційної системи UNIX	42
1.7.2. Взаємодія драйверів з програмним та апаратним середовищем	43
1.7.3. Функція ioctl	45
1.7.4. Структура таблиці bdevsw	46
1.8. Конфігурування ядра	49
Контрольні питання	51
Контрольні вправи	51
Розділ 2. Файлова система	52
2.1. Поняття про звичайні файли	52
2.2. Поняття каталогів	53

2.2.1. Кореневий каталог /	56
2.2.2 Каталог /var	57
2.2.3 Каталог /etc	57
2.2.4 Каталог /usr	57
2.2.5 Каталог /home (/export/home)	58
2.2.6 Каталог /opt (/opt1, /opt2, ... /optn)	59
2.3 Сучасні механізми роботи з оперативною пам'яттю	59
2.3.1 Призначення спеціального розділу SWAP	59
2.3.2 Рекомендації по керуванню розділом swap	59
2.3.3 Команди керування swap розділом	60
2.3.4 Додатковий моніторинг використання swap	61
2.4. Поняття про спеціальні файли	61
2.4.1 Клас пристрою спеціальних файлів	62
2.4.2 Тип і номер спеціальних файлів	63
2.4.3 Зв'язок фізичної і логічної структур дискового простору	63
2.5. Створення файлової системи мережеских ОС	64
2.6. Монтовність і демонтовність файлових систем	65
2.6.1 Дискові пристрої в ОС UNIX	68
Контрольні питання	69
Контрольні вправи	70
2.7. Внутрішня структура мережеских операційних систем	70
2.7.1. Розподіл дискового простору	70
2.7.2. Індексні дескриптори	71
2.8. Системні виклики мережеских операційних систем для роботи з файлами	77
2.8.1. Функція відкриття файла (open)	78
2.8.2 Функція створення файла (create)	80
2.8.3 Функція роботи з дескриптором відкритого файла (dup)	80
2.8.4. Функція читання (read)	81
2.8.5. Функції захоплення процесів (fcntl) та записів (write)	82
2.8.6 Функція знаходження необхідної позиції у файлі (lseek)	83
2.8.7. Функція закриття файла (close)	84
2.8.8. Стандартні бібліотечні функції введення/виведення	85
2.8.9 Потоків функції введення/виведення (getchar, putchar)	85
2.8.10 Бібліотечні функції доступу до файлів	85
Контрольні питання	87
Контрольні вправи	87
2.9. Приклади програмування на Java для роботи з файловими мережескими системами. Файлові об'єкти	88
2.9.1. Функції виведення записів	88
2.9.2. Буферизоване виведення інформації	91
2.9.3. Клас FileOutputStream	91
2.9.4. Функції операцій читання	93
2.9.5. Клас FileInputStream	94
2.9.6. Операції довільного доступу до файлів	95

2.9.7. Методи читання і запису даних до мережевих файлових систем	96
2.10 Мережеві комунікації операційних систем	96
Контрольні питання	99
Контрольні вправи	99
2.11. Загальна характеристика відомих представників мережевих операційних систем	100
Контрольні питання	112
Контрольні вправи	113
Література	114

ВСТУП

Операційна система (ОС) UNIX вважається досить ефективною у роботі з мережевою платформою для будь-якого типу серверу: локального, корпоративного або глобального. Звісно поряд з іншими операційними системами, UNIX має свої переваги та недоліки. Правила, яких дотримуються більшість професіоналів, говорить про те, що будь-який програмний засіб (в тому числі і операційні системи) необхідно використовувати там, де вони будуть найефективнішими у використанні.

З впевненістю можна сказати, що операційна система UNIX є лідером в управлінні локальними, корпоративними та глобальними серверами. На думку багатьох фахівців, завдяки тому, що ОС Windows розробляється однією фірмою Microsoft, темпи її прогресивних наробок будуть повільні ще багато років.

Незважаючи на великий консерватизм у відношенні засобів адміністрування і відновлення версій, UNIX залишається досить потужною мережевою операційною системою.

На користь всебічного вивчення мережевих властивостей ОС UNIX вказує те, що більшість із провідних сучасних інформаційних технологій зародилися та розвинулись саме в її середовищі: TCP/IP, E-Mail, Internet, Intranet, протокол HTTP, мова HTML, глобальна світова павутина Web. Усі ці технології вийшли з лона UNIX.

Становище, яке відбувається зараз в області інформаційних і комп'ютерних технологій, багато фахівців називають революцією. Рушаться, незламні монополії та авторитети, на зміну їм приходять молоді компанії, що стрімко розвиваються. Програмне забезпечення і новітні покоління комп'ютерів захоплюють усе нові і нові сектори ринку, і потік інвестицій, вкладених у сучасні інформаційні технології, стрімко росте. Процесори, запам'ятовувальні пристрої і мікросхеми досягли такого рівня надійності і компактності, що ввійшли в побутову техніку. Програми на мові високого рівня C та Java можуть масштабуватися, тобто виконуватися на будь-якій платформі: від смарт-карти до мейнфрейма.

Мережі комп'ютерів стали вже настільки звичною справою, що без них не проектується жодна система. У цих умовах різко підвищується частка відповідальності системних інтеграторів і системних адміністраторів, що обслуговують потужні корпоративні сервери. Зараз усе більше і більше з'являється Internet провайдерів і обчислювальних центрів. Більш серйозну увагу викликає основна платформа операційної системи Unix – BSD, поширюється Linux. Тому, сьогодні UNIX – є найпотужнішою мережевою операційною системою, яка також здатна працювати з X Windows (графічним UNIX), TCP/IP, Emacs, UUCP, mail і USENET. Адже в сутності своїй UNIX – це система колективного користування, що найповніше відповідає концепціям Internet/Intranet/Extranet.

Тому майбутнє в новітніх інформаційних технологіях вважається за кваліфікованими системними адміністраторами, інтеграторами та мережевими фахівцями.

У попередніх виданнях [1-3] при підготовці навчального посібника перед нами стояли цілі ознайомити студентів з принципами загальної роботи і змістовно описати механізми функціонування ОС UNIX. Цілі даного навчального посібника стоять принципово інші. До найважливіших з них можна віднести:

- мережеві властивості ОС Unix;
- відмінності ОС Unix від інших платформ;
- характерні особливості у роботі та управлінні комунікаційними процесами;
- управління інформаційними ресурсами ОС Unix за допомогою мов високого рівня C та Java;
- управління ресурсами ОС Unix за допомогою стандартних команд;
- загальна характеристика відомих представників мережесих ОС.

Тому в главі 1 "Архітектура сучасної UNIX" подана інформація про структуру та типові функції, що надані користувачу ОС UNIX. Тут стисло описуються:

- ядро системи;
- режим користувача і режим ядра;
- процеси;
- міжпроцесні комунікації;
- багатокористувацький захист;
- свопинг і пейджинг;
- підсистема керування введенням/виведенням.

Одна з найважливіших характерних рис UNIX, – її файлова система, тому важливий розділ 1 обговорює це питання. В главах цього розділу також подані загальні відомості про механізми міжпроцесних комунікацій мережесих файлових систем на прикладі UNIX, наводяться змістовні приклади на мові високого рівня C ефективної взаємодії з цими ресурсами. Велика увага приділяється такому важливому питанню як багатокористувацький захист інформаційних ресурсів.

Звісно існують характерні риси, які відрізняють ОС UNIX від інших мережесих систем, – механізми свопингу і пейджингу та управління процесами введенням/виведенням. З цього приводу в початковому посібнику надається багато ілюстрованого матеріалу та прикладів на мовах C та Shell.

Розділ 2 "Файлова система UNIX" знайомить студентів з організацією файлової системи ОС UNIX і методами, які використовуються для ідентифікації файлів і структури каталогів. Надається поглиблений матеріал для вивчення структури дискового простору. Для закріплення методів функціонування системи введення/виведення пропонуються конкретні приклади роботи на мовах

високого рівня C та Java. Закінчується другий розділ загальною характеристикою відомих представників мережових ОС.

Всі глави навчального посібника проілюстровані конкретними прикладами програмування, кожний з яких перевірений на факультетських серверах Вінницького національного технічного університету та персональних комп'ютерах у реальних умовах.

До всього матеріалу навчального посібника розроблені контрольні питання та вправи, основні елементи яких були ретельно перевірені автором на лекційних та практичних заняттях з 1996 року.

У тексті навчального посібника під терміном „приклад” мається на увазі програмний код, що реалізовує окрему функцію на мові високого рівня C або Java. Термін „програма” позначає цілком працюючий програмний код, який може виконуватись в ОС Unix на мові високого рівня C або Java.

У даному навчальному посібнику розглядаються різні версії UNIX, але перевага віддається стандартам AT&T (UNIX System Release 4.x) та BSD. Більшість прикладів оснований на елементах програмування та реалізації команд, перевірені на діючих системах Linux, FreeBSD, Sun Solaris, SCO OpenServer 5.0. У контрольних прикладах перевага була віддана FreeBSD, Linux та Sun Solaris з деяких причин.

По-перше, операційна система FreeBSD вважається найпотужнішою мережевою ОС у світі.

По-друге, операційні системи FreeBSD та Linux використовуються на міжнародних та корпоративних (факультетських) вузлах Internet Вінницького національного технічного університету і виконують функції розподілення потоків даних, захист інформаційних ресурсів, тощо.

По-третє, Sun – це достатньо відома компанія, що займається тільки UNIX-машинами. Sun ніколи не зраджує сама собі і підтримує одну-єдину ОС UNIX, на відміну від таких, скажемо, компаній, як DEC і HP. Особливістю Sun є те, що як тільки у світі UNIX з'являється що-небудь нове, воно в першу чергу проходить тестування у Sun. Також вважається, що встановлення будь-яких додаткових продуктів у Sun проходить найякісніше.

1. ЗАГАЛЬНІ ВІДОМОСТІ ПРО ОПЕРАЦІЙНУ СИСТЕМУ UNIX

Перші електронно-обчислювальні машини (ЕОМ) випускалися без операційних систем (ОС). Вони управлялися за допомогою набору перемикачів на пульті керування, і на таких машинах, природно, могли працювати тільки інженери вищої кваліфікації. Бажання зменшити залежність від фахівців і наблизити ЕОМ до простого користувача і викликало появу перших ОС.

Будь-яка ОС являє собою набір програм, що управляють ресурсами ЕОМ. Робота цих програм строго координована.

Класична UNIX - це повноцінна мережева ОС. Цей факт означає, що одночасно багато користувачів можуть працювати на одній машині, одночасно виконувати багато програм. У світі мережевих операційних систем серед таких популярних систем, як Windows NT чи OS/2, багато хто з хакерів інтуїтивно розуміють, що UNIX – єдина "реальна річ" – реальна операційна система.

На рис. 1.1 зображено набір сфер, кожна з яких являє собою специфічний UNIX-software, що включає такі компоненти, як:

- ядро;
- Shell (командний інтерпретатор);
- деякі системні утиліти;
- програмні засоби, що використовуються в режимі виконання команд;
- деякі підсистеми верхнього рівня.

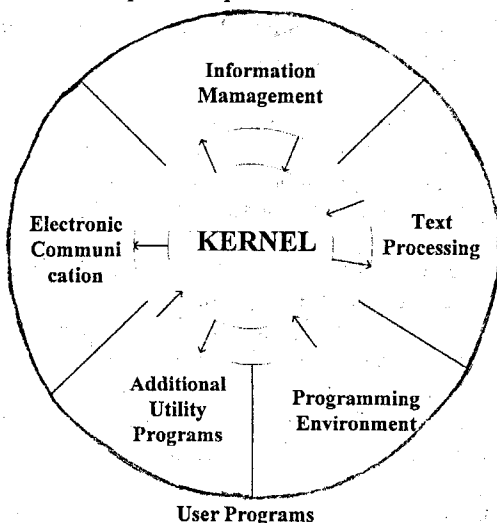


Рис. 1.1. Загальна структура операційної системи Unix

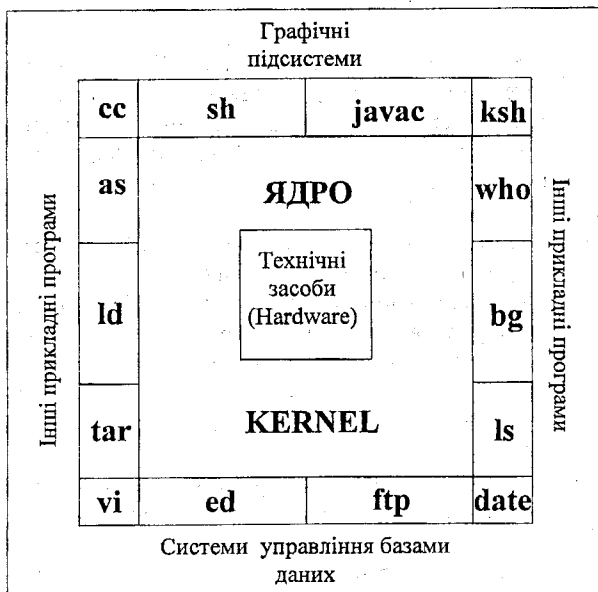


Рис. 1.2. Структура програмного забезпечення UNIX

Для того, щоб зрозуміти як працює UNIX, студент повинен мати уявлення про кожен компонент системи.

1.1. Ядро і процеси

Резидентна частина в оперативній пам'яті UNIX-системи називається ядром (Kernel). При початковому завантаженні (boot) ядро зчитується в пам'ять починаючи з нульової адреси. На диску ядро завжди оформлене як виконуваний файл. При будь-якому перенастроюванні системи, включенні додаткових зовнішніх пристроїв і нових драйверів, зміні параметрів генерації системи ядро перебудовується і зберігається в новому файлі. Як правило, цей файл розташовується в кореневому каталозі і має назву ідентифікатор UNIX-системи (/unix, /hp-ux і т.д.).

Ядро містить системні програми, що виконують диспетчерські функції, керування ресурсами ЕОМ, розподіл пам'яті, доступ до дискових пристроїв і т.і. Умовне ядро ОС UNIX можна представити у вигляді декількох тісно взаємодіючих компонентів (рис. 1.3).

Якщо порівняти ОС UNIX з іншими відомими ОС, то можна знайти, що її ядро мінімізоване і не виконує ні однієї функції, що безпосередньо служить користувачу. Для цієї мети застосовуються численні утиліти, що виступають посередником між користувачем і ядром. У цьому

виявляється архітектурна особливість UNIX, що привертає до неї увагу розробників апаратних архітектур ЕОМ. З цього приводу, головним структурним елементом ОС UNIX є ядро (рис. 1.3).

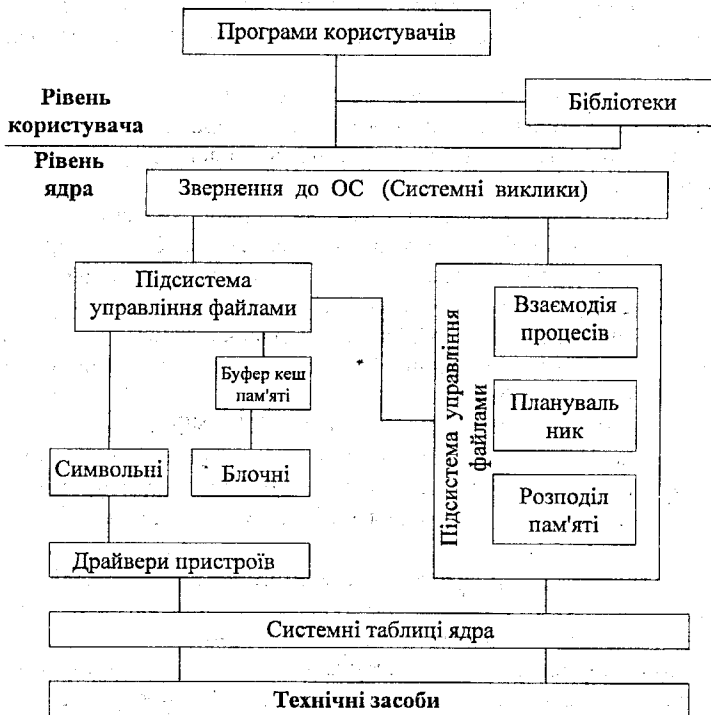


Рис. 1.3. Структура ядра ОС UNIX

Ядро UNIX містить важливі таблиці, у яких знаходиться системна інформація для побудови зв'язаних, координованих програмних ланцюжків. У цьому значенні UNIX – відмінний приклад програми, структура якої визначена даними. При цьому основна частина роботи ядра саме і зводиться до пошуку в таблицях і до їхньої модифікації.

Ядро ОС UNIX підтримує завантаження тільки потрібних сторінок оперативного запам'ятовувального пристрою (ОЗП). Тобто з диска в пам'ять завантажуються ті сегменти програми, що дійсно використовуються. Можливе використання однієї сторінки, фізично один раз завантаженої в пам'ять, декількома виконуваними програмами.

Для збільшення об'єму доступної оперативної пам'яті UNIX здійснює також розбивку диска на сторінки, тобто на диску може бути виділено до 256 Мбайт простору для свопінгу (swap space). У деяких версіях UNIX в область свопінгу вивантажується не весь процес, а тільки окремі його частини, у яких є необхідність виконувати задачу.

Коли системі потрібно більше оперативної пам'яті, то вона за допомогою своінгу виводить неактивні сторінки на диск. Це дозволяє виконувати об'ємніші програми та одночасно обслуговувати в мережі більше користувачів. Однак своінг не виключає нарощування оперативної пам'яті, оскільки він знижує швидкодію, збільшує час доступу.

Ядро також підтримує універсальний пул пам'яті для користувацьких програм і дискового кешу. При цьому для дискового кешу може використовуватися вся пам'ять, і навпаки, кеш зменшується при роботі великих програм.

Програми, що використовують динамічно зв'язані чи розділені бібліотеки, можуть спільно виконувати запуск засобів, які представлені одним фізичним файлом на диску. Це дозволяє файлам займати менше місця на диску, особливо тим, що багаторазово використовують бібліотечні функції. У ОС UNIX також існують бібліотеки, які динамічно зв'язуються під час виконання і дозволяють програмісту замінити бібліотечні модулі своїми власними розробками.

Для забезпечення налагодження ядро ОС UNIX видає дампи пам'яті для подальшого аналізу. Використання дамів пам'яті і динамічних налагоджувачів дозволяє визначити причини непрацездатності програми.

Розглянемо роботу ядра мережевої ОС більш докладно.

1.2. Диспетчер мережевої операційної системи UNIX

Диспетчер ОС UNIX - це програма, що дозволяє процесам конкурувати між собою за час центрального процесора. Мультипрограмування - це квантування часу між групою одночасно виконуваних процесів. Призупинення і поновлення процесів відбувається багато разів у секунду, тому створюється ефект безупинного протікання процесу, тобто одночасне виконання декількох програм. Як ілюстрацію розглянемо приклад на рис. 1.4.

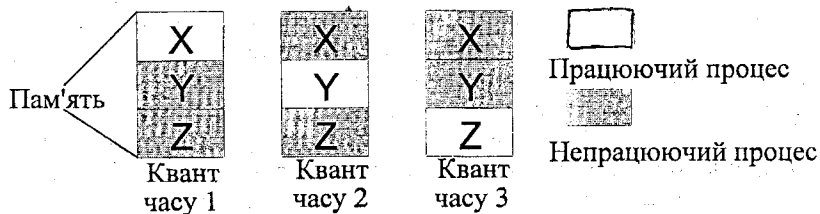


Рис. 1.4. Схема поділу часу центрального процесора мережевої ОС

Наприклад, три програмісти бажають виконати різні програми X, Y, Z. Диспетчер ОС UNIX завантажує ці програми в пам'ять і створює 3 процеси. Різниця терміна "програма" від терміна "процес" у даному випадку визначається тим, що програма - це усього лише файл у форматі

завантаження, що зберігається на диску, а процес розташований у пам'яті на етапі його виконання будь-якого завдання.

В даному випадку, після того як процес Z одержав свій квант часу та обробив його дескриптор, він повертає керування процесу X. Але процес X починає виконання не з самого початку, а з того місця, де той був припинений наприкінці свого кванта часу.

1.3. Процеси мережевої ОС UNIX

Концепція процесів є базовою в ОС UNIX. Саме ядро UNIX існує для забезпечення потреб процесів. Програма в термінах UNIX, як уже говорилося вище, являє собою виконуваний файл. Це основа побудови контексту процесу при його породженні. Тому, породження будь-якого процесу в UNIX – це створення деякої віртуальної машини. Вона має свій власний адресний простір, куди поміщається процедурний сегмент і сегмент даних виконуваної програми.

1.3.1. Дескриптор і контекст процесу

Системні дані використовуються ядром для ідентифікації процесу. Ці дані існують протягом усього часу життя процесу та утворюють дескриптор (описувач) процесу. Множина дескрипторів утворюють таблицю процесів.

Розмір таблиці процесів, хоча і має припустимі обмеження, але в сучасних версіях ОС UNIX дозволяє створювати до декількох сотень дескрипторів які мають певні параметри. Інформація про їх загальний стан включає: розташування (адреса в пам'яті); розмір частини системного образу; номер ідентифікатора процесу та ім'я користувача-власника, який його виконує.

Інша важлива інформація про процес зберігається в таблиці користувача де записуються: ідентифікаційні номери користувача і групи; параметри доступу до файлів; посилання на системну таблицю файлів для усіх відкритих процесом файлів; покажчик на дескриптор поточного каталогу; список реакцій на різні сигнали ОС UNIX.

1.3.2. Ієрархія процесів

Процеси мережевої ОС UNIX утворюють ієрархічні відносини між собою. Існують поняття "процес-батько" і "процес-син". Процеси можуть породжувати інші процеси і їхні відносини найкраще виражає родинне древо. Наприклад, рис. 1.5 показує відносини процесів P1, P2, P3, P4, P5 і P6.

У цьому графі існують такі відносини:

- P1 породжує два процеси: P2 і P4. При цьому P1 є "батьком", а P2 і P4 є його "синами".
- P2 створює P3. При цьому P2 є "батьком" P3, а P3 - відповідно його "сином" і "нащадком" P1.
- P4 є "батьком" P5 і P6; P5 і P6 - відповідно "дітьми" P4.

- P1 є "дідузем" P3, P5 і P6.
- P3, P5 і P6 - "онуки" P1.
- P3, P5 і P6 не мають породжених процесів.

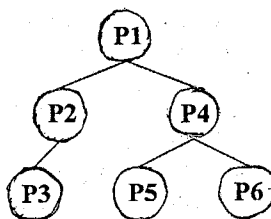


Рис. 1.5. Структурна схема відносин процесів P1, P2, P3, P4, P5 і P6

"Процес-син" успадковує усі властивості "процесу-батька", включаючи змінні оточення, поточні робочі директорії, відкриті файли. Будь-який процес в ОС UNIX породжується за допомогою системного виклику `fork`. Після виконання `fork` обидва процеси продовжують виконання з однієї і тієї ж точки. Для того, щоб довідатися чи є процес "батьком", чи "сином", досить скористатися такою програмою:

```

main ()
{
    int childPID, ParentPID;
    if ((childPID = fork()) == -1 {
        perror ("Can't fork");
        exit(1);
    } else if (childPID == 0) {
        /* процес-син */
        printf ("child: childPID=%d, ParentPID =%d\n",
            getpidQ, getppid());
        exit(0);
    } else {
        /* процес-батько */
        printf ("parent: childPID=%d, ParentPID=%d\n",
            childPID, getpid());
        exit(0);
    }
}
  
```

Приклад 1.1. Визначення типу процесу

Процес, що породжує, може призупинити виконання свого завдання до завершення одного з "процесів – синів" за допомогою системного виклику `wait (&status)`.

Значення, що буде повернуто в змінну `status`, містить у молодшому байті ідентифікатор процесу, що завершився, а в старшому байті – статус завершення. Будь-який процес може завершитися за допомогою

системного виклику exit (status). При цьому, аргумент status передається "процесу-батьку", якщо той очікує завершення "процесу-сина".

ОС UNIX, яка створена як засіб керування процесами, може розглядатися як система рівнобіжних взаємодіючих процесів з деревоподібною структурою. Загальний батько всіх процесів у системі є процес init, що знаходиться у вершині генеалогічного дерева. Цей процес породжується особливим чином і далі постійно є присутнім у системі. Всі інші процеси породжуються в системі за уніфікованою схемою при допомозі системного виклику fork().

1.3.3. Ідентифікатори процесу

Кожному створеному процесу ОС UNIX призначає унікальний ідентифікатор, відомий як процес ID (чи абревіатура PID). Цей PID ідентифікує процес і для самої ОС, і для деяких команд і системних викликів. На додаток до цього, кожен процес має також важливий параметр - parent process ID (абревіатура PPID), що означає PID його "процесу-батька".

Використовуючи команду ps, користувач може побачити ідентифікатори поточних процесів у системі.

Наприклад, якщо ім'я користувача (username) було terry, то на екрані після виконання команди ps можна побачити:

\$ps-f

UID	PID	PPID	3	STIME	TTY	TIME	COMMAND
terry	3865	3699	2	13:35:43	ttyp3	0:00	ps-f
terry	3699	3699	0	12:58:21	ttyp3	0:00	ksh

Цей фрагмент показує, що користувач terry має два процеси: ps-f команду і ksh (Korn Shell). Необхідно відмітити, що PPID процесу ps-f команди такий же, як PID ksh. Це тому, що ps-f був виконаний з ksh-командного рядка; тобто ksh є "процес-батько", а ps-f є "процес-син" ksh.

1.3.4. Групи процесів ОС UNIX

Техніка ідентифікаторів використовується також при групуванні процесів. Кожен процес (за винятком системних процесів, таких, як init і swapper) є членом групи (process group). Коли створюється завдання (JOB), оболонка Shell призначає всім процесам у завданні ідентифікатор групи процесів (GID - process group ID). Сигнали (це спеціальні системні виклики UNIX) можуть поширюватися по всіх процесах у даній групі. Це дуже важлива особливість у керуванні завданнями, коли всі родинні процеси в завданні можуть одержувати сигнали, що їм посилаються.

Наприклад, при встановленні Informix for UNIX, існує особливість, яка полягає у наявності у системі користувача з довільним ім'ям, але з обов'язковою належністю до групи Informix. В багатьох випадках це є неодмінною умовою роботи з системами управління базами даних.

Ще один спосіб групування – це об'єднання процесів у термінальну групу. Члени групи одержують як атрибут ідентифікатор термінальної

групи TGID, що відповідає значенню PID лідера цієї групи. Тому як лідер виступає процес, що відкрив термінал. При цьому, керуючий термінал керування використовується для передачі через процес керування (тобто процес-лідер) сигналів для всіх процесів – членів групи. Сигнали створюються при натисканні визначених клавіш на клавіатурі. Слід визначити, що кожен користувач, що входить у систему за допомогою команди login, обов'язково породжує нові процеси. В цей час кожен процес у контексті користувача має той самий ідентифікатор користувача UID. Відповідність між UID і login-ім'ям користувача знаходиться у файлі /etc/passwd. В цьому випадку, користувачі одержують різні привілеї при яких процес суперкористувача (superuser) має UID та дорівнює нулю. Інші користувачі мають цілі позитивні значення ідентифікаторів, яких більш ніж 1000 (для ОС FreeBSD).

1.3.5. Взаємодія процесів ("батьки", "діти", "сироти", "зомбі")

Відразу після обробки системного виклику fork() відбувається встановлення двостороннього зв'язку між "процесом-батьком" і "процесом-сином". "Син" при цьому успадковує усі властивості "батька". Однак існування нового, породженого процесу як точної копії іншого не має ніякого змісту. Тому майже завжди "син" замінює програмний код "батька" на свій власний програмний код. Це можливо за допомогою системного виклику exec. Наприклад:

```
main ( )
{
    int status;
    if (fork() == 0)
        execl ("/bin/primer", "primer", 0);
    wait (&status);
}
```

Приклад 1.2

На цьому прикладі показано, що "процес-батько" очікує закінчення виконання "процесу-сина", за допомогою системного виклику wait. Але на практиці існують деякі інші випадки завершення процесу. Так, якщо процес створює системний виклик exit, то він стає "процесом-самогубцем" і породжує цілу низку проблем які описані нижче.

Кожен процес може водночас бути як "батьком" та і "сином". Тому, при знищенні процесу можуть залишатися "сироти", долю яких треба вирішувати. В цьому випадку є декілька варіантів:

1. Нормальне закінчення.

"Процес-батько" виконує виклик wait і чекає, коли "процес-син" завершить своє виконання по exit. Однак, якщо в "сина" були свої "процеси-діти", щоб вони не залишилися "сиротами", відбувається їхнє "усиновлення" прабатьком усіх процесів init. Головна робота ядра ОС UNIX при цьому полягає у перевстановленні PPID у "процесів-синів".

Він стає для них рівним одиниці. Крім того, усім процесам, членам групи, посилається відповідний сигнал.

2. Ненормальне закінчення.

"Процес-син" виконує виклик exit в той час коли існує його "батько" який не знаходиться в стані чекання wait. У цьому випадку зв'язок "батька" з "сином" продовжує існувати, незважаючи на те що "син" вже не існує. Зв'язок "процес-син" у таблиці процесів зберігається. Тому такий не до кінця знищений процес називають "зомбі".

3. Передчасний вихід.

"Процес-батько" закінчується раніше своїх "синів". У цьому випадку відбувається реконфігурування генеалогічного дерева і всі "процеси-сироти" перенаправляються постійно діючому процесу init.

1.3.6. "Процеси-демони"

Якщо класифікувати процеси в ОС UNIX, то можна визначити групи:

- користувачеві;
- системі;
- "демони" (daemons).

Системні процеси виконуються в режимі суперкористувача та орієнтовані на виконання системних функцій.

У випадках, коли процесам користувачів потрібно виконати системну функцію, створюється системний виклик. Фактично відбувається виклик ядра системи як підпрограми. З моменту появи системного виклику процес вважається системним. Таким чином, користувачий і системний процеси є двома фазами одного процесу, але вони ніколи не перетинаються між собою. Кожна фаза користується своїм власним стеком. Диспетчерський процес характеризується тим, що не має користувацької фази.

"Процеси-демони" - це визначений різновид системних процесів. "Демони" відрізняються від звичайних процесів тим, що вони виконують специфічні системні дії, наприклад адміністрування і управління в комп'ютерних мережах. Так, типовим "процесом-демоном" є оброблювач вхідних викликів (incoming calls) на UNIX-сервері, що обробляє всі заявки, які надходять з віддаленого з'єднання.

Способи породження і запуску "демонів" можуть бути різні:

1. У процесі старту системи з файлу /etc/rc.

Такі "демони" будуть працювати в статусі суперкористувача протягом всього часу роботи ОС.

2. Використовуючи системний файл /usr/lib/crontab, або користувачий crontab.

Звичайно стандартний системний процес cron протягом дня виконує визначені задачі з файлу /usr/lib/crontab.

3. За допомогою команди at "демон" чекає настання визначеного часу і виконує завдання.

4. При запуску фонового процесу за допомогою користувацького терміналу.

Контрольні питання

1. Загальна характеристика операційної системи Unix.
2. Поняття про ядро та процеси, структура та основні функції. Призначення процесу свопінгу.
3. Диспетчер операційної системи Unix. Поняття про мульти-програмування. Приклад розподілення часу при роботі центрального процесора.
4. Процеси. Загальна характеристика процесів. Поняття про дескриптор та контекст процесу, призначення та функції.
5. Ієрархія процесів. Робота функції системного виклику. Призначення генеалогічного дерева. Програмна реалізація виконання системного виклику (на мові високого рівня C). Відомі функції на мові високого рівня C, які використовуються при виконанні ієрархії процесів.
6. Призначення ідентифікаторів процесів, взаємозв'язок ідентифікаторів з ієрархією процесів. Програмна реалізація групи процесів. Файли, які відповідають за функціонування групи процесів.
7. Взаємодія процесів. Варіанти закінчення процесів, їх характеристика. Приклад нормального завершення процесу (на мові високого рівня C).
8. Поняття про "процеси-демони", їх класифікація, причини виникнення. Відомі способи створення "процесів-демонів", їх характеристика.

Контрольні вправи

1. На основі прикладу 1.1 напишіть текст програми на мові високого рівня C для структурної схеми (рис. 1.5).
2. Виконайте аналіз генеалогічного дерева процесів такого прикладу:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
hoshaba	9192	9191	0	12:22	pts/0	00:00:00	-bash
hoshaba	9231	9192	15	12:22	pts/0	00:00:00	ps -f
hoshaba	9234	9192	15	12:24	pts/0	00:00:00	cat /etc/passwd
hoshaba	9235	9192	13	12:26	pts/0	00:00:00	man date
hoshaba	9237	9192	13	12:29	pts/0	00:00:00	info time
3. Напишіть програму створення каналу на мові високого рівня C.

1.4. Міжпроцесні комунікації

Для побудови будь-яких складних мережових структур в ОС UNIX існують такі потужні засоби:

- сигнали;
- семафори;
- програмні канали;
- черги повідомлень;
- поділювана пам'ять;
- спеціальні команди (write, cu, mail);
- засоби міжмашиної взаємодії (uucp, tcp/ip, nfs, rfs.).

Розглянемо деякі з цих засобів більш докладно.

1.4.1. Сигнали

Якщо розглядати процес як деяку віртуальну машину, то в такій системі повинен існувати механізм переривань, що відповідає стандартним вимогам:

- обробка виняткових ситуацій;
- засоби обробки зовнішніх і внутрішніх переривань;
- засоби керування системою переривань (маскування і демаскування),

Цим вимогам в ОС UNIX відповідає техніка сигналів, що може не тільки сприймати та обробляти сигнали, але і їх породжувати і посылати на інші сервери комп'ютерних мереж.

Сигнали можуть бути синхронними – коли ініціатором сигналу є процес, та асинхронними, коли ініціатор виникнення сигналу – інтерактивний користувач. Джерелом асинхронних сигналів може бути також ядро, коли воно контролює визначені стани апаратури, розглянуті як помилкові.

Для зручності використання кожен сигнал закодований і цифрове значення сигналу з застосуванням мнемонічної форми запису зберігається у файлі <signal.h>.

В таблиці 1.4.1 наведений перелік сигналів представлений у порядку зростання значень сигналів (ціле число, зазначене в дужках).

Таблиця 1.4.1

Імена сигналів	Методи створення та поширення сигналу	Реакція (за замовчуванням)
SIGHUP (1)	Породжується при відключенні зв'язку з терміналом. Поширюється на всі процеси TGID.	Закінчити процес, що прийняв сигнал
SIGINT (2)	Створюється при натисканні визначеної клавіші. Поширюється на всі процеси TGID	Закінчити процес
SIGQUIT (3)	Створюється при натисканні клавіші QUIT. Поширюється на всі процеси TGID.	Те ж

SIGILL (4)	Створюється при виникненні апаратних неконтрольованих станів. Поширюється на поточний PID.	“
SIGTRAP (5)	Створюється при трасуванні. Поширюється на поточний PID.	Закінчити процес
SIGIOT (6)	Створюється при несправності в апаратурі. Поширюється на поточний PID.	Те ж
SIGTEM (7)	Створюється при виконанні команди EMT. Поширюється на поточний PID.	“
SIGFPE (8)	Створюється при обробці чисел із рухомою крапкою. Поширюється на поточний PID.	“
SIGKILL (9)	Створюється при видачі системного виклику kill. Поширюється на адресний PID.	“
SIGBUS (10)	Створюється при помилці в непрямій адресації. Поширюється на поточний PID.	“
SIGSEG (11)	Створюється при виході за межі сегмента. Поширюється на поточний PID.	“
SIGSYS (12)	Створюється при неправильному системному виклику. Поширюється на поточний PID.	“
SIGPIPE (13)	Створюється при записі в Pipe, коли немає процесу зчитування. Поширюється на поточний PID запису в канал.	“
SIGALARM	Створюється при закінченні тимчасової установки. Посилається до PID процесу.	“
SIGTERM (15)	Створюється при виконанні команди kill у командному режимі. Поширюється до адресного PID.	“
SIGUSR (16)	Створюється при системному виклику kill. Поширюється до адресного PID.	“
SIGCLD (18)	Створюється коли закінчується "процес-син". Поширюється до "процесу-батька".	Вилучення сигналу
SIGPWR (19)	Створюється при зменшенні напруги мережі. В залежності від реалізації.	У залежності від реалізації
SIGPOOL (22)	Створюється при реєстрації події в Steam-пристрої. Поширюється до процесу, що працює з потоком.	Закінчити процес

Засобом створення та визначення сигналів в ОС UNIX служать два системних виклики kill і signal.

Системний виклик kill посилає обраному процесу сигнал з визначеним номером:

```
int kill (int pid, int sig);
```

Системний виклик signal сприймає та ідентифікує сигнал:

```
#include <signal.h>
```

```
int (* signal (int sig, void (* func (int))) (int));
```

Можливості системного виклику signal дозволяє обробляти сигнали таким чином:

1. Реакція за замовчуванням.

2. Замаскувати сигнал (тобто проігнорувати його). Для вказівки можливості маскування, другому аргументу присвоюється спеціальне значення SIG_IGN. Можна, наприклад, перевірити чи було раніше встановлене маскування по відношенню до сигналу SIGINT і, якщо не було, то встановити таке маскування:

```
#include <signal.h>
```

```
if (signal(SIGINT, SIG_IGN) != SIG_IGN)
```

```
    signal(SIGINT, SIG_IGN);
```

3. Перехопити посланий процесу сигнал і написати власну підпрограму обробки переривання по асинхронно посланому сигналу.

Наприклад, програма роздруку файлів сприймає сигнал переривання (клавіша DEL) і, призупинивши своє виконання, запитує чи дійсно потрібно перервати друк файла. За допомогою механізму сигналів це може бути реалізоване в такий спосіб:

```
#include <signal.h>
```

```
#include <stdio.h>
```

```
FILE *fildes, *fopen;
```

```
int c;
```

```
int sig;
```

```
main (argc, argv)
```

```
char **argv;
```

```
int argc;
```

```
{
```

```
    extern int myint();
```

```
    int k;
```

```
    if ((k=signal(SIGINT, myint)) < 0) {
```

```
        perror("Програма 1.4.1.1");
```

```
        exit();
```

```
    }
```

```
    argc--;
```

```
    argv++;
```

```
    while(argc--){
```

```
        if (sig == 0) {
```

```
            if ((fildes = fopen(*argv++, "r")) == NULL) {
```

```
                perror("Програма 1.4.1.1");
```

```
                exit(1);
```

```

    }
    while ((c=getc(fildes)) != EOF)
        putchar(c);
    fclose (fildes);
}
}
myint ()
{
    char cl;
    printf ("Продовжити? ");
    cl = getchar ();
    if (cl == 'y')
        sig =1;
    return;
}
exit ();
}

```

Програма 1.4.1.1

Використовуючи техніку сигналів, можна вирішувати найрізноманітніші задачі, у тому числі запобігти таку неприємну особливість, як поява в системі "процесів-зомбі".

В ОС UNIX 4.3 BSD та System V є засіб, побудований на використанні сигналу SIGCLD: signal (SIGCLD, SIG_IGN).

Такий виклик повідомляє ядру, що далі "процес-батько" не зобов'язаний знати долі своїх "синів". Тому ядро буде само закінчувати процеси по exit або іншим способом і "зомбі" з'являться не будуть.

1.4.2. Семафори

Семафори поряд із програмними каналами, чергами повідомлень і поділюваною пам'яттю входять у так звану групу IPC – засоби міжпроцесних комунікацій (Interprocess Communication).

Семафори – це досить могутній засіб синхронізації процесів.

Насамперед, у будь-якій ОС є критичні ресурси, з якими у певний відрізок часу може працювати тільки один процес, що захоплює ресурс. Тому семафори за визначеною схемою керують доступом до критичних ресурсів. Самі по собі вони мають значення цілих чисел, з якими припустимі тільки дві атомарні (неподільні) операції.

Семафори – це засоби затримки процесів, що гальмуються чи "засинають" на світлофорах та "пробуджуються" і починають рухатися при зміні стану семафорів. Таким чином дескриптор семафора є інформаційною структурою керування ipc-perm і зберігається у файлі <sys/ipc.h>.

Процес може створити семафор за допомогою системного виклику `semget`. В результаті успішного виконання `semget` повертається `semid` - ідентифікатор створеного чи вже наявного семафора.

Після створення семафора кожний процес, що його знає ідентифікатор `semid`, може на нього впливати за допомогою системного виклику `semctl`, що здійснює налаштування і перевірку стану множинного семафора.

Семафор є також засобом реєстрації деякої події (подій) у системі. Про настання таких подій повідомляють процеси, використовуючи системний виклик `semop`, що приводить до перевстановлення поточних значень семафора.

Процес, що виконує системний виклик `semop`, повинен використовувати поле `sem_op` у структурі `sembuf`, в якому задається необхідна операція:

```
struct sembuf {
```

```
    ushort sem_num; /* номер елемента семафора */
```

```
    short sem_op; /* конкретна операція над елементом семафора */
```

```
    short sem_flg /* прапори */
```

Розрізняють 3 типи операцій з семафорами:

1. `sem_op > 0` - звільнення процесу, що виконав `semop` та ресурсу, який був контрольований даним елементом семафора. Поточне значення елемента семафора `semval` інкрементується на величину `sem_op`.

2. `sem_op = 0` - процес, що виконує такий виклик `semop`, буде переведений у очікування на час, поки не звільниться ресурс, тобто поки `semval` не стане рівним нулю.

3. `sem_op < 0` - процес, що викликав виконання цієї операції, чекає коли `sem_op` стане більшим чи рівним `semval`. У даному випадку відбувається декримент поточного значення семафора на абсолютну величину `sem_op`. Ця операція використовується при захопленні ресурсу.

На семафорах припустимо застосовувати спеціальні прапори, наприклад `IPC_NOWAIT`, що інформує ядро про небажання процесу переходити в стан чекання, незважаючи на "червоне" світло семафора. Це продиктовано прагненням уникнути небезпеки блокування всіх процесів, які знаходяться у семафорі. Це відбувається у випадку, якщо з будь-якої причини процес, що захопив ресурс, раптом аварійно або достроково завершився після одержання сигналу `kill`. Оскільки сигнал `kill` неможливо перехопити системою, то настає ситуація неможливості звільнення семафора і ресурсу. В цьому випадку необхідно використовувати сигнал `SEM_UNDO`.

1.4.3. Програмні канали

Програмні канали являють собою засіб обміну потоками інформації між процесами. Це засіб синхронізації процесів, тому що різні процеси завжди можуть коректно взаємодіяти між собою. Спроба запису в канал, який ніхто не читає, приведе до призупинення "процесу-письменника" і

появи "процесу-читача". Спроба читання з каналу, у який ніхто не пише, приводить до припинення "процесу-читача" і появи "процесу-письменника".

Механізм програмних каналів існував в ОС UNIX з моменту народження. Однак, в останніх версіях багатьох різновидів UNIX з'явилося багато потужних особливостей.

По-перше, знята проблема взаємодії (як по сигналах, так і по програмних каналах) тільки для "процесів-родичів", що принципово обмежувало спільність засобів взаємодії процесів.

По-друге, у додатку до `pipe` з'явилися нові засоби у вигляді черг повідомлень і поділюваних областей пам'яті.

В основі `pipe` - це деякий спеціальний файл, у який можна писати інформацію чи читати з нього. Це укладається в загальну концепцію введення/виведення в ОС UNIX. Вибірка і розміщення інформації в такий особливий файл повинна відбуватися за технологією FIFO (First In/First Out).

При введенні/виведенні завжди використовується потокова модель даних. Це означає, що передані через `pipe` дані ніяк не інтерпретуються. Довжина переданих повідомлень не контролюється. Відразу декілька "процесів-клієнтів" можуть писати в канал, а один процес-сервер може писати з нього та обмінюватися з "клієнтами" повідомленнями. При цьому повідомлення ніяк не відокремлюються один від одного і сервер сам повинен визначати, хто з "клієнтів" і яке повідомлення надіслав до нього.

Існує два види програмних каналів:

- неіменовані програмні канали;
- іменовані програмні канали.

Неіменовані канали характеризуються тим, що вони можуть встановлюватися тільки між процесами, що породжені одним вихідним предком, який створив канал. Породжені процеси отримують у спадщину дескриптор відкритого файлу і можуть писати та зчитувати інформацію з каналу.

Іменований канал надає користувачам ширші можливості. Після створення такого каналу його можуть експлуатувати всі процеси. Для цього існує ім'я каналу, що повинно бути відоме всім процесам, які збираються брати участь в обміні. Ім'я каналу не отримують у спадщину, як дескриптори відкритих файлів. Воно встановлюється один раз, відразу після створення каналу і після цього стає доступним усім. Створити іменований програмний канал можна за допомогою команд ОС Unix, виконавши такі дії:

```
mknod name p
```

Після цього можна записати що-небудь до каналу, наприклад:

```
cat proba.txt > p
```

Потім, зареєструйтеся на будь-якому підключеному до UNIX терміналі:

login: username
password: password

Після того, як користувач отримає підказку ОС для введення команд, наприклад - \$, необхідно ввести таку команду:

cat p

На екрані термінала з'явиться інформація, яка попередньо була записана до каналу.

Зазвичай програмний канал створюється за допомогою системного виклику `pipe`, що повертає два дескриптори файлу: один для запису в канал, інший для читання з каналу. Після цього всі операції передачі даних здійснюються за допомогою системних викликів типу `read/write`. Програмний канал можна зобразити у вигляді труби, що працює за схемою FIFO: першим прийшов – першим обслужили (рис. 1.6)

Після створення програмного каналу і початку роботи з ним система введення/виведення забезпечує припинення процесів, якщо канал заповнений інформацією (за замовчуванням його обсяг 4096 байт), тобто при записі операція `write` успішно проходить, якщо в `pipe` є вільний простір, чи переходить у стан чекання, якщо канал містить інформацію. В процесі читання усе відбувається навпаки – операція `read` зупиняється, якщо канал порожній, або успішно завершується, якщо в каналі є інформація.

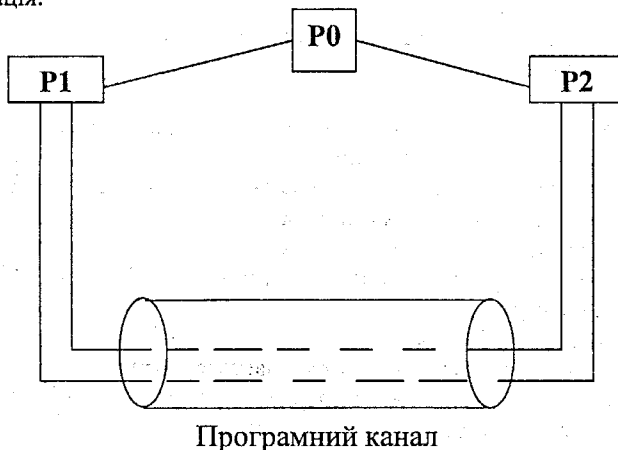


Рис. 1.6. Структурна схема програмного каналу

В ОС Unix кожен процес може встановити декілька програмних каналів, так що можна організувати проходження даних у вигляді лінійних чи мережевих структур (рис. 1.7.).

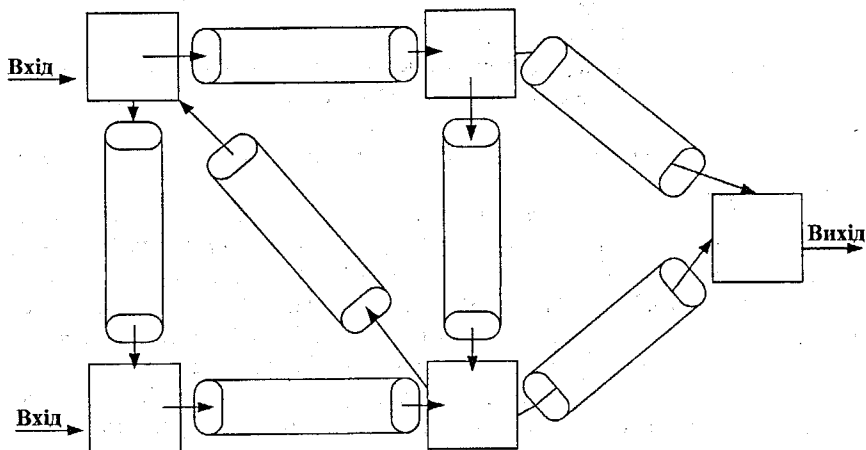


Рис. 1.7. Структурна схема мережевої структури програмних каналів

При використанні цього засобу необхідно розглянути дві особливості:

1. Процес запитує через `read` більший обсяг даних, ніж існує в каналі. В результаті цього процесу буде передано тільки фактичну кількість переданих даних.

2. Якщо при відкритті файла був використаний сигнал `ON_DELAY` (в іменованому каналі), то при виявленні порожнього каналу процес, що видає `read`, не переходить в стан чекання. Йому відразу передається керування після `read` зі значенням рівним нулю.

Часто конвеєрна обробка команд ОС Unix реалізується з використанням програмних каналів лінійної структури. Командна послідовність вигляду:

```
proc1 | proc2 | proc3
```

розглядається оболонкою Shell яка складається з трьох його "процесів-синів": `proc1`, `proc2`, `proc3`. В міру їхнього породження Shell установлює програмні канали від `proc1` до `proc2` і від `proc2` до `proc3`.

Як ілюстрацію приведемо два приклади використання програмних каналів.

Програма 1.4.3.1

Вихідна програма виконує паралельний процес, створює програмний канал і передає в нього вміст файла `proba.txt`. Породжений процес закриває свій файл стандартного введення і замінює його на програмний канал. Після цього виконується стандартна програма друкування файлів `cat`, що читає дані з програмного каналу:

```
#include <stdio.h>
int pipel[2];
```

```

char cbuf[80];
int i,j;
int err,n;
it fd;
main ()
{
    if((err=pipe(pipel) < 0)) { /* Створення каналу */
        perror ("помилка роботи програми 1.4.3.1");
        exit();
    }
    j = fork();                /* Створення нового процесу */
    if (j) {
        if((fd = open("proba.txt",0)) < 0) {
            perror("файл proba.txt не знайдений");
            exit ();
        }
        while ((n = read(fd,cbuf,80)) > 0) /* відкриття файла для запису */
            write(pipel[1],cbuf,n);
        close(fd);
        close(pipel[1]); /* Закриття каналу */
        exit();          /* Закінчення "процесу-сина" */
    }
    close(0);
    fd=dup(pipel[0]);
    close(pipel[0]);
    close(pipel[1]);
    execir/bin/cat", "cat", 0);
    perror("помилка роботи програми 1.4.3.1");
}

```

Програма 1.4.3.2

Реалізувати модель типу “клієнт – сервер” з використанням іменованих програмних каналів. Такі канали можуть бути засобом комунікації між незалежними процесами. У цьому прикладі використовуються дві незалежні програми – одна для клієнта, інша для сервера.

Програма-сервер:

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
#define chan1 "/tmp/fifo.1"
#define chan2 "/tmp/fifo.2"
#define PERM 0666

```

```
#define MAXBUFF 1024
```

```
main ()
```

```
{
```

```
    int fdr, fdw;
```

```
    /* Створити іменовані програмні канали */
```

```
    if ((mknod(chan1, S_IFIFO | PERM, 0) < 0) &&  
        (errno != EEXIST))
```

```
        printf("Не можу створити канал 1: %s, chan1");
```

```
    if ((mknod(chan2, S_IFIFO | PERM, 0) < 0) &&  
        (errno != EEXIST)) {
```

```
        unlink(chan1);
```

```
        printf("Не можу створити канал 2: %s, chan2");
```

```
    }
```

```
    /* Відкрити іменовані програмні канали */
```

```
    /* chan 1 - на читання, chan2 - на запис */
```

```
        if ((fdr=open(chan1,0)) < 0)
```

```
            printf("\\"Процес-батько\":" не можу відкрити канал для  
читання: %s", chan1);
```

```
        if ((fdw=open(chan2,1)) < 0)
```

```
            printf("\\"Процес-батько\":" не можу відкрити канал для запису:
```

```
%s", chan2);
```

```
        server(fdr,fdw);
```

```
        close (fdr);
```

```
    close (fdw);
```

```
}
```

```
server(fdr,fdw)
```

```
int fdr,fdw;
```

```
{
```

```
    char buff[MAXBUFF];
```

```
    char err[256];
```

```
    int n, fd;
```

```
    extern int errno;
```

```
    printf("Програма-сервер");
```

```
    if ((n=read(fdr, buuf, MAXBUFF)) <= 0)
```

```
        perror("Програма-сервер: помилка читання файла ");
```

```
        buff[n] = '\0';
```

```
        if ((fd=open(buff,0)) < 0) {
```

```
            /* відправлення повідомлення від програми-сервера до програми-  
клієнта */
```

```
            sprintf(err, "Помилка: не можу відкрити файл %s\n");
```

```
            strcat(buff,err);
```

```
            n=strlen(buff);
```

```
            if (write(fdw,buff,n) != n)
```

```
                perror("Програма-сервер: помилка запису файла ");
```

```

} else {
n=0;
while ((n=read(fd,buff,MAXBUFF)) > 0) {
    if (write(fdw,buff,n) != n)
        perror("Програма-сервер: помилка запису файла");
    if (n < 0)
        perror("Програма-сервер: помилка запису файла");
}
}

```

Програма 1.4.3.2 реалізує серверну частину по створенню іменованого каналу.

Програма-клієнт:

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
#define chan1 "/tmp/fifo.1"
#define chan2 "/tmp/fifo.2"
#define PERM 0666
#define MAXBUFF 1024
main ()
{
    int fdr, fdw;
    /* Відкрити іменовані канали.
    /* Ці канали повинні бути підготовлені сервером */
    if ((fdw = open(chan1, 1)) < 0)
        printf("Програма-клієнт: не можу відкрити канал для запису: %s ",
               chan1);
    if ((fdr = open(chan2, 0)) < 0)
        printf("Програма-клієнт: не можу відкрити канал для читання: %s: %s",
               chan1);
    client(fdr, fdw);
    close(fdr);
    close(fdw);
    if (unlink(chan1) < 0)
        printf("Програма-клієнт: не можу виконати операцію зі зв'язком %s",
               chan1);
    if (unlink(chan2) < 0)
        printf("Програма-клієнт: не можу виконати операцію зі зв'язком %s",
               chan2);
    exit(0);
}

```

```

client (fdr,fdw)
int fdw,fdr;
{
    char buff[MAXBUFF];
    int n;
    printf("Програма-клієнт");
    if (fgets(buff,MAXBUFF,stdin)==NULL)
        /*Прочитати назву каналу */
        perror("Програма-клієнт: помилка читання імені
        файла");
    n=strlen(buff);
    f (buff[n-1] == '\n')
        n--;
    f (write(fdw, buff, n) != n)
        perror("client: filename write error");
    /* читання даних з каналу */
    while ((n = read(fdr,buff,MAXBUFF)) > 0 )
    if (write(1, buff, n) != n) /* type on standard I/O */
        perror("помилка запису даних");
        if (n < 0)
            perrorf("Програма-клієнт: помилка читання даних");
    }

```

Програма 1.4.3.3 реалізує клієнтську частину по створенню іменованого каналу.

Для демонстрації цього прикладу спочатку необхідно відтранслювати обидві програми:

```
cc -o client client.c
```

```
cc -o server server.c
```

і потім виконати програму-сервер у фоновому режимі:

```
server &
```

Після цього виконати клієнтську програму:

```
client
```

В результаті роботи цих програм буде створений програмний канал, який приведе до обміну інформацією між сервером та клієнтом. У випадку неможливості виконання цих програм буде створено повідомлення про наявність помилки.

1.4.4. Черги повідомлень

Черги повідомлень також є засобом обміну інформацією між процесами, але на відміну від програмних каналів припускають більш гнучку організацію взаємодій процесів.

В цьому випадку зовсім не обов'язкове існування парних процесів "читача" і "письменника". Просто буде існувати деяка черга повідомлень, що використовується як сховище повідомлень для будь-яких процесів, які до неї звертаються, - і "читачі" і "письменники".

При формуванні черги повідомлень не використовується потокова модель даних. Кожне повідомлення має строго визначену структуру: тип і дані. У повідомленні припустимі будь-які дані. Довжина даних вибирається користувачем і може бути довільною в межах пам'яті, відведеної під розміщення черги в момент її створення.

Ядро ніяк не інтерпретує зміст повідомлень, а тільки забезпечує розміщення і вибірку відповідно до зазначеного типу. Шаблон повідомлення зберігається у файлі <sys/msg.h>.

```
struct msgbuf {  
    long mtype;    /* тип повідомлення */  
    char mtext[1] /* дані */  
};
```

Для роботи з чергами в ОС UNIX існують такі виклики:

- msgget – створення черги;
- msgctl – установка параметрів;
- msgsnd – посилка повідомлень у чергу;
- msgrcv – одержання повідомлень з черги.

Черги мають свої переваги в порівнянні із програмними каналами. Наприклад, щоб здійснювати повнодуплексний зв'язок між двома процесами потрібно створити два канали: один для запису, а інший для читання. У випадку використання черг можна обійтися однією чергою, якщо розрізнити повідомлення за типом: від клієнта до сервера і навпаки.

Крім того, є можливість забирати повідомлення з черги не тільки по дисципліні FIFO, але й у довільному порядку. Ця особливість відкриває перспективу побудови систем із пріоритетним обслуговуванням клієнтів.

1.4.5. Розподілена пам'ять

Розподілена пам'ять є іншою важливою особливістю комунікацій між процесами. Для того щоб збільшити швидкість проходження даних між процесами застосовують розподілену пам'ять. В усіх розглянутих вище випадках обмін інформацією між процесами проходив через ядро. Техніка розподіленої пам'яті надає процесам інший, простий, але дуже ефективний спосіб обміну інформацією, не через ядро, а використовуючи деяку частину віртуального адресного простору в який вони поміщають і звідки зчитують дані.

Зрозуміло, ця ідея не нова, але реалізація її в ОС UNIX пропонує зручні засоби для програмістів

Після створення розподіленого сегменту пам'яті, кожний з "процесів користувачів" може приєднати його до свого власного віртуального простору і працювати з ним, як зі звичайним сегментом пам'яті. Єдиним недоліком при цьому є відсутність засобів синхронізації. Однак при цьому можна використовувати, наприклад, техніку семафорів.

Для розподілених сегментів використовуються такі системні виклики і header-файли:

```
#include <sys/types.h>
```

#include <sys/ipc.h>

shmget - створення сегмента;
shmetl - установка параметрів;
shmat - приєднання сегмента;
shmdt - від'єднання сегмента.

Більш детальну інформацію з цього приводу можна знайти у джерелах [1-5] наведеної літератури.

Контрольні питання

1. Загальна характеристика міжпроцесних комунікацій. Відомі сервіси підтримки моделі „клієнт-сервер” за допомогою міжпроцесних комунікацій. Історія розвитку міжпроцесних комунікацій.

2. Сигнали, основна характеристика, властивості роботи, призначення. Синхронні та асинхронні сигнали. Приклади та характеристика відомих сигналів.

3. Методи обробки сигналів за допомогою функції signal. Наведіть приклад.

4. Призначення та основні функції семафорів. Відомі операції з семафорами.

5. Програмні канали, призначення та основні характеристики. Методи роботи з іменованими та неіменованими програмними каналами. Створення каналів за допомогою команд ОС Unix та функцій мови високого рівня C.

6. Призначення черг повідомлень. Основні властивості команд у ОС Unix з управління чергами повідомлень. Програмна реалізація черг повідомлень.

7. Поділювана пам'ять. Методи роботи. Команди ОС Unix по управлінню поділюваною пам'яттю. Програмна реалізація поділюваної пам'яті.

Контрольні вправи

1. Напишіть програму на мові високого рівня C, яка усуває виникнення „процесів-демонів” за допомогою механізму сигналів.

2. За допомогою команд операційної системи Linux або FreeBSD створіть програмний канал та запишіть в нього інформацію з файла /etc/fstab. З іншої консолі спробуйте прочитати інформацію з каналу.

3. Виконайте завдання 2 використовуючи одну системну консоль.

4. За допомогою команд операційної системи Linux або FreeBSD напишіть приклад по структурі:

```
proc1 | proc2 | proc3  
proc1 | proc2 > proc3  
proc1 < proc2 > proc3
```

5. Назвіть основну відмінність у створенні програмних каналів приклади 1 та 2.

6. Побудуйте блок-схему алгоритму програм для прикладів 1 та 2. Опишіть основні блоки.

1.5. Багатокористувачевий захист

Оскільки ОС UNIX є не тільки багатозавданевою, але і мережевою ОС, фахівцями питання захисту інформаційних ресурсів завжди приділялася достатня увага. Для цього існують потужні механізми багатокористувачевого захисту до яких відносяться ідентифікатори користувача і групи, ефективні та реальні ідентифікатори, привілеї користувачів.

1.5.1. Ідентифікатори користувача і групи

Параметри користувачів, яким дозволена робота в системі, відображаються у файлі користувачів `/etc/passwd`. Параметри користувачів, які об'єднані в групи, знаходяться у файлі `/etc/group`. Кожному користувачу призначається цілочислений ідентифікатор і пароль.

Коли користувач входить у систему, виконується пошук рядка в обліковому файлі `/etc/passwd`. Якщо поле пароля порожнє, пароль не записується.

З кожним процесом зв'язані два ідентифікатори: ідентифікатор користувача та групи.

З кожним файлом також зв'язані два ідентифікатори: користувача і групи. Файл успадковує їх від процесу, що його створив. Користувач і група, ідентифікатори яких зв'язані з файлом, вважаються його власниками. Можна змінити власника файлу за допомогою команди `chown`, та змінити групу – за допомогою команди `chgrp`. Ідентифікатори файла визначають права доступу до нього. Можуть існувати 3 категорії:

- власник файлу (процес, у якого ідентифікатор користувача збігається з ідентифікатором власника файлу);
- члени групи (процеси, у яких ідентифікатор групи збігається з ідентифікатором групи, до якої файл належить);
- інші групи – всі інші процеси.

Процеси можуть мати різні способи доступу до файлу, наприклад: читання, запис, виконання.

У кожного файлу є код захисту, який при створенні був йому присвоєний. Код захисту – це параметр в індексному дескрипторі файлу з встановленими одиничними значеннями бітів:

000400 - читання власником;

000200 - запис власником;

000100 - виконання дій власником;

000040 - читання членами групи;

000020 - запис членами групи;

000010 - виконання дій членами групи;
000004 - читання іншими користувачами;
000002 - запис іншого користувача;
000001 - виконання іншим користувачем.

При спробі доступу до файлу в процесі виконання будь-яких дій визначається категорія, у яку він попадає стосовно привілей. З коду захисту вибираються 3 біти, що повинні відповідати цій категорії. Якщо збігу не відбулося, тобто жоден біт не відповідає виконанню заданої операції, ядро відкидає запит процесу. Зрозуміло, можна змінити код захисту будь-якого файлу, але зробити це може тільки привілейований користувач або власник конкретного файлу чи каталогу.

1.5.2. Привілейований користувач

Процес, ідентифікатор користувача якого дорівнює нулю, вважається привілейованим. Користувач, щовикористовує цей ідентифікатор, називається привілейованим.

Незалежно від коду захисту файлу, привілейований користувач має доступ до усіх файлів у системі. Звичайно в обліковому файлі користувачів у `/etc/passwd` є привілейований користувач з ім'ям `root`. Як правило, його застосовує системний адміністратор, відповідальний за функціонування системи.

1.5.3. Ефективні і реальні ідентифікатори

У параметрах захисту файлу передбачені спеціальні біти, що дозволяють змінити ідентифікатори користувача і групи: 004000 - дозвіл зміни ідентифікатора користувача при виклику файлу, 002000 - дозвіл зміни ідентифікатора групи. Це пов'язано з механізмами захисту системи.

Тому, початкові ознаки, які пов'язані з процесом, називаються реальними ідентифікаторами. Ідентифікатори, які отримані процесом після виконання системного виклику `exec`, — ефективними. Ознаки можуть збігатися, якщо в коді захисту файлу передбачена зміна ідентифікаторів. Це відбувається у випадках коли завантаження файлу виконується по функції `exec`. Тоді реальні ідентифікатори процесу змінюються на ідентифікатори власника (чи групи) виконуваного файлу, тобто стають ефективними.

Права доступу процесу перевіряються по його ефективних ідентифікаторах. Наприклад, власником файлів даних `dat1`, `dat2`, `dat3` є користувач з ідентифікатором 1423. Будь-який доступ до цих файлів дозволений тільки їхньому власнику. Крім того, є виконувані файли `prog1`, `prog2`, `prog3`, власником яких є той же самий користувач. Потрібно, щоб інші процеси, які намагаються робити операції над файлами `dat1`, `dat2`, `dat3`, могли це робити тільки за допомогою програм `prog1`, `prog2`, `prog3`. Для досягнення цієї мети в коді захисту файлів `prog1`, `prog2`, `prog3` повинна бути дозволена зміна ідентифікатора користувача

для процесу, що викликав один з цих файлів. Тоді будь-який інший процес, якому необхідно працювати з одним з файлів `dat1`, `dat2` чи `dat3`, повинний викликати по `exes` відповідну програму обробки `prog1`, `prog2` чи `prog3`. Інакше він узагалі не одержить доступу до цих захищених власником файлів даних `dat1`, `dat2`, `dat3`.

Також кожен процес може системними викликами `getuid`, `getgid`, `getegid` довідатися про свої реальні та ефективні ідентифікатори. Процес може динамічно змінювати свої ідентифікатори системними викликами `setuid`, `setgid`, однак це дозволено тільки привілейованому користувачу або такому процесу, у якого реальний ідентифікатор збігається з установленим.

Спеціальна програма `passwd` дозволяє змінити пароль користувача в обліковому файлі користувачів `/etc/passwd`. Власником командного (`passwd`) або текстового (`/etc/passwd`) файлів є користувач з ім'ям `root`. Код захисту облікового файла `/etc/passwd` дозволяє записувати в нього тільки власнику, а код захисту файла, що виконується, `passwd` дозволяє зміну ідентифікатора користувача.

Однак користувач, який не є системним адміністратором, може змінити свій пароль тільки за допомогою програми `passwd`. Тим самим дотримується коректність стосовно користувача, який завжди може змінити свій пароль не повідомляючи про це системного адміністратора.

1.6. Знайомство з механізмом свопінгу та пейджингу

1.6.1. Загальний механізм роботи

Механізм свопінгу в ОС UNIX полягає в тому, що у випадку, якщо на комп'ютері малої потужності існує недостатня кількість оперативної пам'яті, то описана в підрозділі 1.2 схема функціонування ОС у частині диспетчеризації і керування процесами працює ефективно. Але в будь-який період часу оперативна пам'ять комп'ютера (яка є обмежена) може повністю заповнитися працюючими процесами. Тому після того як один з користувачів побажає виконати нову програму, диспетчеру ОС необхідно знайти спосіб помістити її в пам'ять. Це приведе до виконання механізму свопінгу, тобто переміщення образів процесів між оперативною і дисковою пам'яттю.

Цей механізм полягає в тому, що коли пам'яті не вистачає і необхідно виконати нову програму, диспетчер ОС копіює один із процесів у пам'яті на жорсткий диск. Після цього диспетчер ОС завантажує нову програму (створює новий процес) і надає цій програмі можливість виконати програмний код. Пізніше процес, скопійований на диск, обмінюється місцями з одним із процесів у пам'яті (виконує свопінг). Так процес з пам'яті копіюється на диск, а процес з диска – назад у пам'ять.

Уявимо собі, що існують 3 процеси: а, b, і с і з'являється заявка від користувача на виконання програми d (рис. 1.8.).

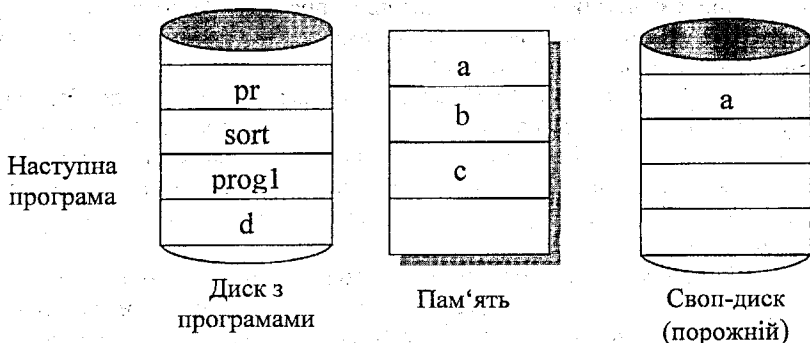


Рис. 1.8. Реалізація нової програми d

Оскільки в пам'яті немає місця для процесу d, процес а копіюється на диск (рис. 1.9.).

Після того як процес а буде скопійований на диск, копія d буде завантажена в пам'ять на місце а (рис. 1.10.).

Після декількох квантів часу диспетчер ОС повертає процес а назад у пам'ять і, як правило, заміняє ним у пам'яті той процес, що знаходився там довше усіх. Припустимо, що процес а обмінюється місцями з процесом b. Перше, що потрібно зробити, це скопіювати процес b на диск (рис. 1.11.).

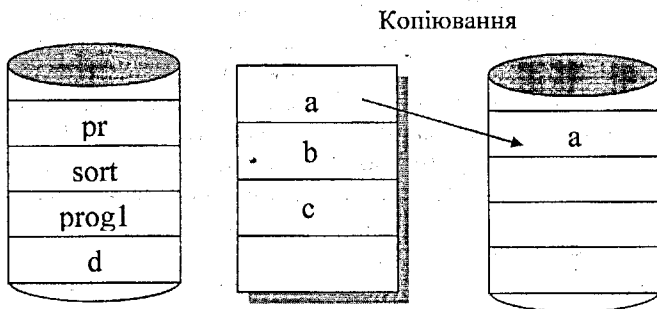


Рис. 1.9. Реалізація копіювання а на своп-диск

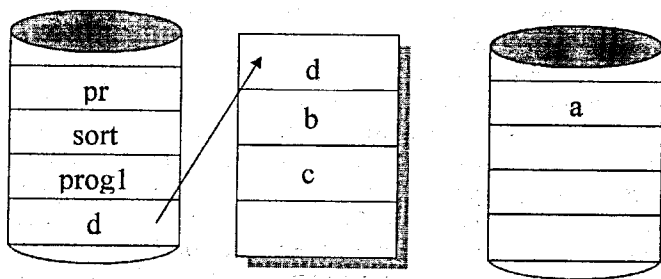


Рис. 1.10. Реалізація копіювання d у пам'ять

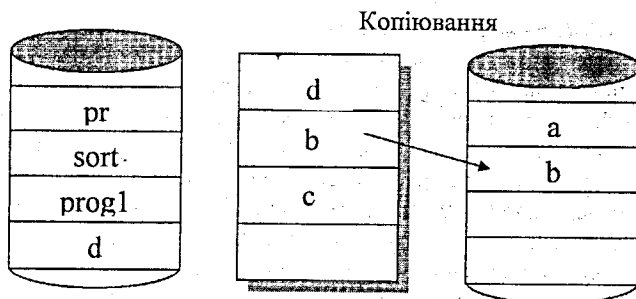


Рис. 1.11. Реалізація копіювання b на своп-диск

Після цього a копіюється назад у пам'ять (рис. 1.12.).

Цей метод копіювання процесів між диском і пам'яттю продовжується доти, поки не буде досить місця для всіх процесів, розташованих одночасно в пам'яті. Зрозуміло, це тільки примітивна ілюстрація дійсної роботи диспетчера ОС UNIX.

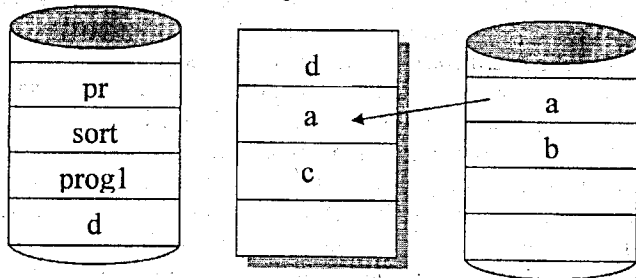


Рис. 1.12. Реалізація копіювання a в пам'ять

В даний час більшість UNIX-систем замість свопінгу використовують механізм, названий пейджингом. Він подібний свопінгу, за винятком того, що на диск копіюються частини програм (сторінки). Сторінки звичайно мають фіксований розмір по 2048 або 4096 байт. Коли виникає потреба в пам'яті, одна зі сторінок копіюється на диск, звільняючи порцію пам'яті. Пізніше, коли ця сторінка необхідна виконуваний програмі, вона переноситься назад у пам'ять, можливо витісняючи на диск яку-небудь іншу сторінку. Незважаючи на те, що пейджинг складніший свопінгу, він працює більш ефективно, тому що переносяться тільки малі частини програм. Крім того, якщо велика програма використовує тільки малу частину пам'яті протягом часу свого виконання, то лише найчастіше використані порції програмного коду будуть постійно присутні в дорогій головній пам'яті.

1.6.2. Буфер кеш-пам'яті

Ядро ОС UNIX завжди працює з файлами, розміщеними на зовнішніх запам'ятовувальних пристроях великої ємності дискової пам'яті. Це дозволяє процесам зберігати нову порцію інформації або викликати раніше збережені дані. Якщо процесу необхідно звернутися до файла, то ядро вибирає інформацію в оперативну пам'ять де процес зможе переглядати ці дані, змінювати їх і звертатися з вимогами про їх повторне збереження у файлової системи.

Звісно, ядро могло б робити читання і запис безпосередньо з диска і на диск при всіх звертаннях до файлової системи. Однак при цьому час реакції системи і продуктивність були б низькими через повільну швидкість передачі даних з диска. З цієї причини ядро намагається звести до мінімуму загальну частоту звертань до диска, створивши спеціальну область внутрішніх інформаційних буферів. Цю структуру називають буферним кешем. Це не слід плутати з апаратним кешем, що прискорює непряму адресацію пам'яті.

Перед обробником інформації з диска ядро завжди намагається зчитати що-небудь з буфера кешу. Якщо в цьому буфері відсутня інформація, то ядро читає дані з диска. Потім заносить їх у буфер, використовуючи алгоритм, що має на меті помістити в буфері якнайбільше необхідних даних. Аналогічно інформація, яка записується на диск, заноситься в буфер для того, щоб знаходитися там, якщо ядро пізніше спробує її зчитати. Ядро також намагається звести до мінімуму частоту виконання операцій запису на диск, з'ясовуючи чи повинна інформація дійсно запам'ятовуватися на диску, чи це проміжні дані, які будуть незабаром знищені. Алгоритми обробки інформації вищого рівня дозволяють робити попереднє занесення даних у буфер кешу або затримувати запис даних для того, щоб з максимальною ефективністю використовувати буфер.

Алгоритм виділення буфера повинний бути надійним. Процеси не повинні "засинати" назавжди, тому що рано чи пізно їм потрібно

виділити буфер. Ядро гарантує такий стан при якому всі процеси, що очікують виділення буфера, продовжать своє виконання. В цьому випадку ядро розподіляє буфери під час обробки звертань до ОС і звільняє їх перед поверненням для керування процесами. У режимі задачі процеси безпосередньо не контролюють розподілення буферів, тому вони не можуть навмисно "захоплювати" буфери. Виключенням є системна операція mount, що захоплює буфер доти, поки не буде виконана операція umount. Це виключення не є істотним, оскільки загальна кількість буферів набагато перевищує число активних монтованих файлових систем.

Ядро втрачає контроль над буфером тільки тоді, коли чекає завершення операції введення/виведення між буфером і диском. Звичайно в ОС UNIX, навіть якщо дисковод зіпсований, він не може переривати роботу центрального процесора, але тоді ядро ніколи не звільнить буфер. В цьому випадку дисковод повинний стежити за роботою апаратних засобів і повертати ядру код помилки, повідомляючи про погану роботу диска. Тому ядро може гарантувати, що процеси, припинені для очікування обробки даних у буфері з часом відновлять своє виконання.

Можна також уявити собі ситуацію, коли процес припиняється очікуючи доступу до буфера. У випадку, коли кілька процесів припиняються, очікуючи звільнення буфера, ядро не гарантує, що вони одержать доступ до буфера в тій черзі, у якій вони запросили доступ. Процес може призупинити і відновити своє виконання, коли буфер стане вільним, лише з тією метою, щоб призупинитися знову через те, що інший процес першим одержав керування над буфером. Теоретично так може продовжуватися вічно, але практично такої проблеми не виникає в зв'язку з тим, що в системі звичайно закладена велика кількість буферів.

1.6.3. Переваги і недоліки буферного кешу

Використання буферного кешу має переваги та недоліки.

Насамперед, використання буферів дозволяє однаково виконувати процедуру звертання до диска, оскільки ядру потрібна інформація про технологію введення/виведення. Замість цього ядро копіює дані в буфер і з буфера незважаючи на те, чи є дані частиною файла, чи індексу суперблока. Буферизація введення/виведення з диска підвищує модульність розробки програм, оскільки ті складові частини ядра, що займаються введенням/виведенням на диск, мають один інтерфейс на усі випадки. Таким чином спрощується проектування системи.

Система не накладає ніяких обмежень на вирівнювання інформації користувальницькими процесами, що виконують введення/виведення, оскільки ядро робить внутрішнє вирівнювання інформації. У різних апаратних реалізаціях часто потрібно вирівнювати інформацію для введення/виведення з диска певним чином, наприклад до двобайтового чи чотирибайтового вирівнювання даних у пам'яті. З цієї причини на

машинах з обмеженими технічними можливостями у вирівнюванні адрес виникає велика кількість помилок програмування. Крім того, з'являється проблема переносу програм в ОС UNIX.

Завдяки використанню буферного кеша скорочується обсяг дискового трафіка і підвищується загальна продуктивність системи. Процеси, що зчитують дані з файлової системи, можуть знайти інформаційні блоки в кеші.

Ядро часто застосовує "відкладений запис", щоб уникнути зайвих звертань до диска, залишаючи блок даних у буферному кеші. Проте кількість буферів, які можна закласти в системі, обмежується обсягом пам'яті, що доступна виконуваним процесам. Якщо під буфери звільнити занадто багато пам'яті, то система буде працювати повільніше в зв'язку з тим, що їй прийдеться займатися підкачуванням і заміщенням виконуваних процесів. Сучасні алгоритми буферизації допомагають підтримувати цілісність файлової системи, оскільки вони зберігають загальний початковий і єдиний образ дискових блоків, що містяться в кеші. Якщо два процеси одночасно спробують звернутися до одного і того ж дискового блоку, то алгоритми буферизації (наприклад, `getblk`) паралельний доступ перетворять у послідовний, тим самим запобігаючи руйнуванню даних.

Скорочення дискового трафіка є важливою перевагою з погляду забезпечення якісної продуктивності або швидкої реакції системи. Однак загальна стратегія кешування також має деякі недоліки. Перш за все – це властивість ядра у випадку відкладеного запису не переписувати дані на диск негайно. Така система вразлива для збоїв. Наприклад, при раптовому відключенні живлення дані залишаються диску в некоректному вигляді.

Використання буферного кешу вимагає додаткового копіювання інформації при її зчитуванні і записі користувацькими процесами. Процес, що записує дані, передає їх ядру, і ядро копіює дані на диск. Процес, що зчитує дані, одержує їх від ядра, що читає дані з диска. При передачі великої кількості даних додаткове копіювання негативно позначається на продуктивності системи. Однак при передачі невеликих обсягів даних продуктивність підвищується, оскільки ядро їх буферизує (використовуючи алгоритм `getblk` і відкладений запис) доти, поки це є ефективним з погляду економії часу роботи системи з диском.

Контрольні питання

1. Загальна характеристика ідентифікаторів користувача групи. Поняття про код захисту файла, приклади.
2. Привілейований користувач та адміністратор, їх основні обов'язки.
3. Ефективні та реальні ідентифікатори, призначення та основні функції.

4. Свопінг та пейджинг, загальні характеристики, основні властивості, відмінності у призначенні. Приклад процесу свопінгу.

5. Поняття про буфер кеш-пам'яті. Наведіть загальну характеристику процесів, які використовують буфер кеш-пам'яті. Поясніть ситуації, які можуть виникати під час виконання процесів у кеш-пам'яті.

6. Переваги та недоліки буферного кешу.

Контрольні виправи

1. В ОС Unix знайдіть файли паролів та ідентифікаторів груп. Визначте привілеї та охарактеризуйте кожного з користувачів.

2. Знайдіть параметри одиничних значень бітів для файлів паролів та ідентифікаторів груп. Поясніть значення параметрів для кожної з груп одиничних бітів.

3. Побудуйте структурну схему процесу свопінгу для восьми процесів. Поясніть основні етапи.

4. Побудуйте структурну схему процесу пейджингу для чотирьох процесів. Поясніть основні етапи.

1.7. Керування введенням/виведенням

Система керування введенням/виведенням у ОС UNIX базується на концепції драйверів.

Драйвер пристрою – це програма, що забезпечує зв'язок ядра з периферійними пристроями, такими як накопичувачі на магнітних дисках і стрічках, термінали, принтери і мережні пристрої. Тому, в мережних ОС існує поняття – "тип пристрою". Цей тип звичайно однозначно відповідає драйверу. Один драйвер керує безліччю фізичних пристроїв даного типу. Наприклад, один термінальний драйвер може керувати всіма терміналами, підключеними до системи. Драйвер розрізняє пристрої якими керує система. Вихідні дані, призначені для одного терміналу, не повинні бути послані на інший.

ОС також підтримує "псевдопристрої", з кожним з яких не пов'язаний жоден конкретний фізичний пристрій. В ОС UNIX все можна трактувати як пристрій, навіть фізичну пам'ять. Наприклад /dev/mem. З цього прикладу бачимо, що пам'ять не є ніяким пристроєм і ніколи не керується драйвером. Проте, використовуючи цей псевдопристрій, можна звертатися до інформаційних структур ядра у фізичній пам'яті.

Драйвери пристроїв є частиною ядра, а не користувацькими процесами. Однак доступ до драйвера можливий як з ядра, так і з області пам'яті, виділеної користувацькому процесу. Доступ до пристроїв на користувацькому рівні забезпечується за допомогою спеціальних файлів пристроїв, що знаходяться в каталозі /dev. Ядро перетворює операції, виконувані над цими спеціальними файлами, у функції звертання до коду драйвера.

1.7.1. Спеціальні файли операційної системи UNIX

В ОС UNIX крім звичайних файлів і каталогів існують спеціальні файли. Їх можна використовувати для позначення фізичних пристроїв комп'ютера і псевдопристроїв. Будь-яку послідовність байтів у ОС UNIX можна розглядати як файл. Користувач цілком може вводити і виводити у файл `/dev/console`, що відповідає консольному терміналу. Все, що виводиться у файл `/dev/console`, буде з'являтися на екрані терміналу. Все, що вводиться з клавіатури, можна довідатися в результаті читання цього файла. Наприклад:

```
su
echo Hello! > /dev/console
cat /dev/console
CTRL+C
```

Приклад 1.7.1.1

Для кожного терміналу UNIX-машини існує відповідний йому файл. Звичайним терміналам відповідають файли, що прописуються приблизно так:

`/dev/contty01` чи `/dev/tty02` чи `/dev/ttyFD02` чи `/dev/tty0` (і т.і.)

Ці файли відносяться до псевдотерміналів і характерні тим, що створюються під час роботи користувача в системі у віддаленому режимі за допомогою команд типу `ssh`, `telnet`. За їх роботу відповідають також спеціальні файли `/dev/pty*`, чи `/dev/pty/*`. Наприклад, за допомогою команд `who` або `who am i` можна одержати назву власного терміналу. Існують також спеціальні файли, що співвідносяться з жорстким диском (`/dev/rdisk/c0t3d0s2`) або його розділами (`/dev/dsk/c0t3d0s`).

За допомогою стандартних утиліт ОС UNIX, можна розмістити жорсткий диск на велику кількість частин і кожній з них надати відповідне ім'я (наприклад, за допомогою програми `divvy` у SCO UNIX). На кожній частині можна змонтувати окрему файлову систему чи використовувати її як виділений дисковий простір для спеціальних цілей: `свопінг` або `пейджинг`. Кожній іменованій частині на дисковому поділі усередині UNIX Partition буде відповідати спеціальний файл. Якщо користувач буде читати з цих файлів, то одержить всі байти в тому порядку як вони розташовані на жорсткому диску: з нульового циліндра і до останнього. У випадку, якщо користувач спробує що-небудь записати в цей файл, то втратить дискову розмітку і безнадійно зіпсує систему.

Крім спеціальних файлів, що описують конкретні пристрої (термінали, диски і т.д.), у ОС Unix існують інші спеціальні файли, такі як:

`/dev/mem` файл, що відображає оперативну пам'ять;
`/dev/zero` нульових пристрій для тестування процесу виведення інформації;
`/dev/audio` файл, що відображає звуковий пристрій;

/dev/null нульових пристрій для тестування процесу введення/виведення інформації.

Спеціальні файли поділяються на два типи:

- блокові - запис і читання допускаються тільки блоками. Наприклад, поділи дисків з файловою системою. Наприклад, якщо виконати команду:

```
ls -al /dev/fd*
```

```
brw-rw--- 1 root floppy      2, 28 Jul 18 1997 /dev/fd0H1440
```

```
brw-rw--- 1 root floppy      2, 32 Jul 18 1997 /dev/fd0H2880
```

```
brw-rw--- 1 root floppy      2, 12 Jul 18 1997 /dev/fd0H360,
```

то можна визначити файли з атрибутом "b";

- символічні (безструктурні дані) - можна читати і писати окремими байтами. До них відносяться термінали, магнітні стрічки, неформатовані диски і розділи дисків без файлової системи. Наприклад, якщо виконати команду:

```
$ls -al /dev/tty*
```

```
crw--w--w- 1 sbd   sys  4,1   Feb 24 11:00 /dev/tty1
```

```
crw--w--w- 1 root  root  4,2   Feb 24 20:40 /dev/tty2
```

```
crw--w--w- 1 root  root  4,2   Feb 24 20:40 /dev/tty2
```

то можна визначити файли з атрибутом "c".

Однак, поняття "розмір" у спеціального файла не існує. Адже цей файл визначає фізичний пристрій. Фактично це посилання на відповідний драйвер. Замість довжини команда `ls` показує для таких файлів два числа: "major" і "minor" які означають номери пристроїв: major - це порядковий номер драйвера пристрою, а minor - внутрішній номер пристрою в таблиці драйвера, що його обслуговує.

1.7.2. Взаємодія драйверів з програмним та апаратним середовищем

Як уже відзначалося у ОС UNIX є два класи пристроїв – блокові пристрої і пристрої посимвольного введення/виведення. Файлова структура може бути створена тільки для блок-орієнтованих пристроїв.

Блокові пристрої представлені як запам'ятовувальні пристрої з довільним доступом. До пристроїв посимвольного введення/виведення відносяться всі інші пристрої в тому числі термінали і мережне устаткування.

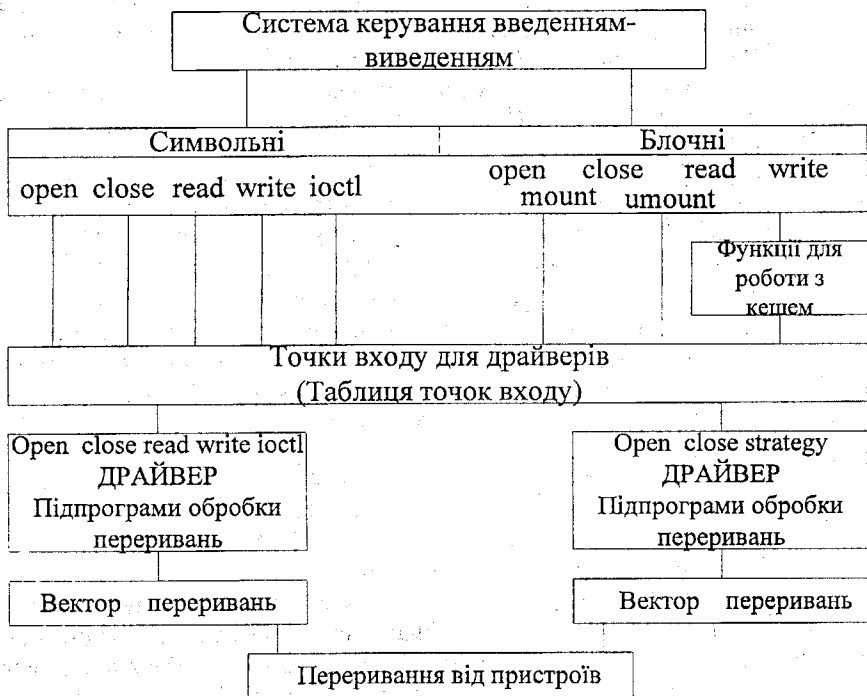


Рис. 1.13. Структура системи керування введенням/виведенням в мережевих операційних системах

Користувач взаємодіє з пристроями за допомогою файлової системи і драйверів (рис. 1.13.). Кожен пристрій має ім'я у каталозі /dev і користувач звертається до нього як до файла. Спеціальний файл пристрою відрізняється від інших файлів типом, що зберігається в його індексі. Він може мати змінні "блоковий", або "символьний спеціальний", у залежності від пристрою, що цей файл представляє. Якщо пристрій має як блоковий, так і символьний інтерфейс, то його представляють два файли: спеціальний файл пристрою введення/виведення блоками і спеціальний файл пристрою посимвольного введення/виведення. Системні функції обробки звичайних файлів, такі, як open, close, read і write, можна застосовувати і для пристроїв. Однак системну функцію ioctl, що керує пристроями посимвольного введення/виведення, не можна застосовувати до файлів звичайного типу. Навпаки, системна функція fcntl забезпечує контроль над діями, які проводяться на рівні дескриптора файла, але не на рівні пристрою. В інших реалізаціях функція ioctl застосовується для файлів усіх типів. Проте, драйверам пристроїв немає необхідності підтримувати повний набір системних функцій.

Байт-орієнтований інтерфейс менш структурований, тому програмування драйверів введення/виведення для таких зовнішніх пристроїв значно простіше.

Інтерфейс системи з блок-орієнтованими пристроями у більшості випадків визначається набором спеціальних функцій і буферизацією блоків пристрою. Тому їхні драйвери повинні задовольняти багато вимог і системних угод. Наприклад, у ядрі використовується процедура *strategy* для передачі даних між буферним кешем і пристроєм. Для пристроїв посимвольного введення/виведення процедура *strategy* існує для безпосередньої передачі даних між пристроєм і адресним простором задачі. Процедура *strategy* також може керувати черговістю виконання завдань на введення/виведення або виконувати більш складні дії. В цьому випадку ядро передає адресу заголовка буфера процедурі драйвера в заголовку якого міститься список адрес (сторінок пам'яті) і розміри даних, переданих на пристрій чи з пристрою. Аналогічна дія має місце і при роботі механізму свопінгу або пейджингу. Якщо дані копіюються в адресний простір задачі, драйвер повинний блокувати процес у пам'яті до завершення передачі даних. Наприклад, після монтування файлової системи ядро ідентифікує кожен файл по номеру пристрою і номеру індексу. У номері пристрою закодовані його старший і молодший номери. Коли ядро звертається до блоку, що належать файлу, воно копіює номер пристрою і блоку в заголовок буфера. Звертання до диска, що використовують алгоритми роботи з буферним кешем (наприклад, *bread* чи *bwrite*), ініціюють виконання процедури, обумовленої старшим номером пристрою. Ця процедура використовує значення полів молодшого номера і номера блоку з заголовка буфера для ідентифікації місця розташування даних на пристрої. Адреса буфера – це ідентифікація місця призначення переданих даних. Також коли процес звертається до пристрою введення/виведення блоками безпосередньо (наприклад, відкриваючи пристрій і читаючи з нього або записуючи в нього), він використовує алгоритми роботи з буферним кешем.

1.7.3. Функція *ioctl*

Системна функція *ioctl* є специфічною функцією і має загальну точку входу для всіх пов'язаних з типом пристрою команд, дозволяючи процесам задавати програмні (асоційовані з драйвером) та апаратні (асоційовані з пристроєм) параметри. Спеціальні дії, які виконують функцію *ioctl*, для різних пристроїв різні і визначаються типом драйвера. Програми, що використовують виклик *ioctl*, повинні знати з файлом якого типу вони працюють. Тому програми, що використовують виклик *ioctl*, є апаратно-залежними.

Взагалі, функція *ioctl* має синтаксис:

```
ioctl(fd,command,arg);
```

де *fd* - дескриптор файла, що повертає попередньо викликану функцією *open*;

command - дія (команда), яку необхідно виконати драйверу;
arg - параметр команди (може бути покажчиком на структуру).

Існують команди які специфічні для різних драйверів. Тому кожний драйвер інтерпретує команди у відповідності зі своїми внутрішніми специфікаціями. Від команди, у свою чергу, залежить формат структури даних, описуваної переданим параметром. Драйвери можуть зчитувати структуру даних arg із простору задачі відповідно до визначеного формату чи записувати установку пристрою в простір задачі за адресою зазначеної структури. Наприклад, наявність інтерфейсу, наданого користувачам функцією ioctl, дає їм можливість установлювати для термінала швидкість передачі інформації в ботах, перемотувати магнітну стрічку і, нарешті, виконувати мережні операції при яких задаються номери віртуальних каналів і IP адреси.

Доступ системи до драйвера відповідного пристрою здійснюється через дві системні таблиці cdevsw і bdevsw, що містять покажчики на підпрограми драйверів (cdevsw – для байт-орієнтованих і bdevsw – для блок-орієнтованих пристроїв):

```
extern struct cdevsw {  
    int (*d_open) ();  
    int (*d_close) ();  
    int (*d_read) ();  
    char *(*d_write) ();  
    int (*d_ioctl) ();  
    int (*d_stop) ();  
    struct tty (*d_ttys);  
} cdevsw [];
```

Структура таблиці cdevsw

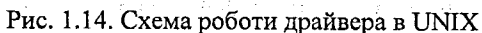
```
extern struct bdevsw {  
    int (*d_open) ();  
    int (*d_close) ();  
    int (*d_strategy) ();  
    struct buf (*d_tab);  
} bdevsw [];
```

1.7.4. Структура таблиці bdevsw

Драйвери байт-орієнтованих пристроїв доступні через таблицю cdevsw у якій знаходяться покажчики для виконання функцій open, close, read, write і ioctl.

Функція ioctl породжується системними викликами stty і gtty. При виклику будь-якої функції драйвера в якості аргументу передаються тип і номер пристрою. Це дозволяє програмі правильно вибрати необхідний контролер з декількох однотипних пристроїв. Також в програмі драйвера є кілька стандартних процедур, серед яких найважливіші - читання (read), запис (write) і керування (ioctl). Точки входу в процедури write і read

Функція `write` тільки записує дані з області даних користувача, але не передає їх на зовнішній пристрій. Для більшості байт-орієнтованих пристроїв є спеціальна функція `crass` для читання байта з області даних користувача. Отриманий байт, як правило, ставиться в чергу. Реальна передача байтів на зовнішній пристрій здійснюється по перериваннях від пристрою в спеціальній програмі обробки даних у драйвері. Таким чином, ці дві процедури працюють асинхронно по відношенню одна до одної та в різних контекстах.



Системний рівень є продовженням (системною фазою) звичайного користувацького процесу, який стандартним шляхом звернувся до пристрою. Підпрограма обробки переривань працює тільки в момент обробки даних, які ініційовані пристроєм. У віртуальному адресному просторі ядра в цей момент може відображатися контекст зовсім іншого процесу. Однак, підпрограми рівня переривань не мають права звертатися до контексту системного рівня і змінювати його поля. Єдина структура, до якої дозволений доступ підпрограмам обробки переривань – це черга байтів, створена системним рівнем. Існують спеціальні функції для роботи з чергами: `putc` і `getc`. Розмір черги, природно, безмежний. Варто враховувати, що буферний простір ядра розділяється всіма активними процесами. Тому визначається максимальна границя. Після заповнення черги процес припиняється по функції `sleep`. Для продовження припиненого процесу по функції `wakeup` необхідно щоб черга була оброблена до деякого значення часу. Так, для занадто повільно діючих пристроїв підпрограми системного рівня можуть заповнювати чергу швидше, ніж пристрій очищає свій регістр даних і генерує переривання. Навпаки, швидкодіючі пристрої можуть мати невеликий розмір черги. Більш того, у ОС UNIX функції `sleep` і `wakeup` використовують як аргументи адресу таблиці відповідного пристрою. Оскільки черга знаходиться в адресному просторі ядра в області буферизації даних, це дозволяє вивантажувати процес під час операцій введення/виведення.

Аналогічним способом працюють і підпрограми читання даних. У різних реалізаціях ОС UNIX можуть бути деякі відмінності в назвах функцій і способі передачі параметрів, але алгоритм обробки даних залишається незмінним.

Що стосується блок-орієнтованих пристроїв, то в 80-х роках існувала проблема коли треба було для більшості апаратних пристроїв мати інтерфейсну плату і замовлений драйвер. В наш час всі проблеми вирішуються завдяки прийняттю в ОС UNIX стандарту SCSI (Small Computer Systems Interface). Стандарт SCSI визначає загальний канал передачі даних, що можуть використовувати периферійні пристрої усіх видів (диски, стрічки, сканери, принтери).

Стандарт SCSI не визначає конкретну конструкцію підключеного до нього пристрою, а лише регламентує взаємодію з іншими пристроями. Існує стандартний роз'єм Centronics або DB-50, за допомогою якого до плоского кабеля SCSI можна підключити вісім або більше пристроїв.

В ОС UNIX після підключення нових пристроїв до комп'ютера, він самостійно їх розпізнає після початкового завантаження і запускає програму `autoboot` для створення спеціальних файлів.

Для SunSparc є спеціальна процедура визначення підключених на SCSI зовнішніх пристроїв. Для цього необхідно натиснути на системній консолі дві клавіші: `Ctrl+A`. Після натискання цих клавіш викликається

ббудований boot-monitor, якому треба дати команду prob-scsi у відповідь на його запит ОК.

Ok> prob-scsi

Сконфігуровані зовнішні пристрої можна подивитися за допомогою команд prtconf, prtvtoc, dkinfo.

1.8. Конфігурування ядра

Задання конфігурації системи – це процедура вказівки системними адміністраторами значень параметрів, за допомогою яких виконується налаштування системи. Деякі з параметрів указують розміри таблиць ядра, таких, як таблиці процесів, індексів і файлів. За допомогою інших параметрів вказується конфігурація пристроїв, тобто ядром виконуються конкретні правила. Також ядро бере на себе функції визначення адрес та пристроїв, які існують в даній системі.

Існують різні ситуації, при яких ядро може бути переконафігуроване. Наприклад, у SCO-UNIX ядро перебудовується при кожній модернізації системи, зокрема при додаванні нового пристрою чи при інсталяції нового програмного засобу, що змінює системні параметри. Системні адміністратори можуть кодувати інформацію про конфігурацію у файлах, що транслуються і компонуються під час побудови ядра. Інформація про конфігурацію пристроїв вказується в простому форматі, де програма конфігурації перетворює її у файл, готовий для трансляції. У деяких системах ядро самоідентифікується внутрішні таблиці конфігурації в момент коли динамічно коректує пристрої. Це надає ядру можливість довідатися, які з пристроїв включені. Також ядро зчитує апаратні ключі для самоналаштування.

В ОС UNIX типу BSD і в деяких версіях Linux при додаванні нового пристрою і його драйвера приходить перекомпільовувати ядро. В разі необхідності можна створювати файл пристрою по команді mknod.

Всі файли можливих пристроїв вже існують у каталозі /dev. Перекомпіляція, як правило, здійснюється в каталозі /usr/src/linux. Спочатку здійснюється побудова нового конфігураційного файла, а потім - компіляція ядра. Більш докладно цей процес міститься у /usr/src/linux/README.

У SunOS.4.1.3 та BSD конфігураційні файли ядра знаходяться в каталозі /sys. За допомогою команди man confg можна отримати повний опис послідовних дій для створення нового ядра.

У Solaris.2.4 (UNIX - SVR4) перекомпільовувати ядро не треба. Досить створити порожній файл /reconfigure по команді:

touch /reconfigure

Потім зупинити комп'ютер командою halt, приєднати новий пристрій і завантажити ОС знову. В цьому випадку перекомпіляція не потрібна, тому що ядро Solaris.2.4 має можливість самогенерації (ядро має тип Generic). Оскільки такі периферійні пристрої як стример, HDD, CDROM,

floppy-disk підключаються через SCSI-шину, то для коректної роботи кожному типу в ОС Solaris.2.4 призначено конкретну SCSI-адресу.

В багатьох випадках якщо виробники поставляють драйвер пристрою, то для його конфігурації використовується команда `drvconfig`.

Інтерфейс "ядро - драйвер" описується в таблицях ключів пристроїв введення/виведення.

Кожен тип пристрою має в таблиці точки входу, які при виконанні системних функцій адресують ядро до відповідного драйвера.

Функції `open` і `close`, які викликаються файлом пристрою, обробляються через таблиці ключів пристроїв відповідно до типу файла. Функції `mount` і `umount` також викликають виконання процедур відкриття і закриття пристроїв, але для пристроїв введення/виведення блоками даних. Функції `read` і `write`, які викликані пристроями введення/виведення блоками даних і файлами в змонтованих файлових системах, виконують алгоритми роботи з буферним кешем. Це ініціює реалізацію процедури роботи з пристроями. Деякі з драйверів виконують процедури читання і запису.

Інтерфейс "апаратура - драйвер" складається з машинно-залежних регістрів або команд введення/виведення для керування пристроями і векторами переривань. Коли відбувається переривання від пристрою, система ідентифікує пристрій, що викликав переривання, і запускає програму обробки відповідного переривання.

Системні адміністратори встановлюють спеціальні файли пристроїв командою `mknode`, у якій указується тип файла (блоковий чи символьний), старший і молодший номери пристрою. Команда `mknode` виконує системну функцію з тим же ім'ям, що створює файл пристрою. Наприклад, у командному рядку:

```
mknode /dev/tty7 c 2 7
```

`/dev/ttyl3` - ім'я файла пристрою;

`c` - указує, що тип файла "символьний спеціальний" (`b`, відповідно, блоковий);

`2` - старший номер пристрою; *

`7` - молодший номер пристрою.

Старший номер пристрою показує його тип, якому відповідає точка входу в таблиці ключів пристроїв; молодший номер пристрою - це порядковий номер одиниці будь-якого типу. Якщо процес відкриває спеціальний блоковий файл з ім'ям `/dev/dsk1` і кодом `0`, ядро виконує програму `gdopen` у точці `0` таблиці ключів пристроїв блокового введення/виведення.

З одним старшим номером пристрою може бути зв'язано безліч периферійних пристроїв. Молодший номер пристрою дозволяє відрізнити їх одне від одного. Для цього не потрібно створювати спеціальні файли пристроїв при кожному завантаженні системи. Необхідно тільки коректувати параметр системи у разі якщо змінилася конфігурація системного блоку. Наприклад, якщо до встановленої

конфігурації були додані пристрої, система їх коректно встановить за допомогою визначених параметрів ядра.

Контрольні питання

1. Загальні характеристики драйверів та пристроїв операційної системи Unix. Поняття псевдопристрою. Відмінності драйверів та пристроїв в операційній системі Unix від ОС DOS або Windows. Місце драйверів та пристроїв в структурі ОС Unix.

2. Використання спеціальних файлів для роботи у фізичних пристроях та псевдопристроях.

3. Різновиди спеціальних файлів, їх особливості. Основні відмінності спеціальних файлів від звичайних.

4. Особливості створення файлової системи. Опишіть загальну структуру системи керування введенням/виведенням в ОС Unix.

5. Загальна характеристика системної функції `ioctl`. Синтаксис, структура, особливості використання.

6. Структура таблиці `bdevsw`. Загальна схема роботи драйверів в ОС Unix, описання особливості роботи.

7. Особливості конфігурації ядра в різних клонах ОС Unix. Необхідність у переконфігурації ядра. Приклади та призначення команд, які допомагають у переконфігурації ядра. Методи отримання додаткової інформації.

8. Поясніть дію кожного рядка прикладу 1.7.1.1

Контрольні вправи

1. Знайдіть каталоги, де розміщені спеціальні файли. Яким чином можна доказати, що ці файли – спеціальні? Опишіть знайдені спеціальні файли та їх призначення.

2. Напишіть програму та наведіть приклади використання системної функції `ioctl`.

3. В наданій викладачем ОС знайдіть та ознайомтесь з інформацією про переконфігурацію ядра.

4. Виконайте переконфігурацію ядра системи.

5. Виконайте зупинку, запуск та перезавантаження системи.

2. ФАЙЛОВА СИСТЕМА

Сучасні версії ОС UNIX можуть підтримувати кілька різних файлових систем, включаючи стандартні файлові системи: AT&T, UNIX Berkeley, Xenix і NFS. У ОС UNIX System V, Release 4.2 додатково включені файлові системи bfs, sfs, ufs, s5 які рекомендовані до використання на процесорах типу Intel 386/486/Pentium. Перед появою ОС System V блоки на диску були стандартного розміру в 512 байт. Далі у версії 4.x ОС System з'являються файлові системи які підтримують різні розміри блоку для прискорення доступу до файлів. Це значно зменшило втрати розміру диска від невикористаних блоків. Подібні методи стали вважатись революційними і знайшли своє використання в інших ОС Unix.

Розглянемо стандартну файлову систему ОС UNIX.

Основною функцією мережевої файлової системи є розподіл дискового простору на іменовані ділянки – файли. Деякі ОС, наприклад VAX/RMS, підтримують різнотипні файли з відповідними методами доступу (прямий, індексний, індексно-послідовний і т.і.). Це значно ускладнює роботу ОС. Але у більшості ОС UNIX файлова система надзвичайно проста, де файли являють собою прості послідовності байтів. Завжди до файлів звертаються як до текстових чи двійкових даних, але розрізняються вони лише змістом, а не структурою і методом доступу. У сучасних умовах цей механізм доступу до файлів не вважається недоліком. Тому ця система універсальна. В цьому випадку не робиться ніяких припущень про внутрішню структуру даних файла. Доступ до будь-якого зовнішнього пристрою, а також до іншого процесу здійснюється як до звичайного файла.

В результаті цього, тимчасові характеристики файлової системи визначаються швидкістю накопичувачів на жорсткому диску, а застосування методів кешування в сполученні з попереднім читанням не викликаних блоків файлів і використанням відкладеного запису дозволяє досить ефективно обробляти файли.

Для користувачів, що працювали з MS DOS, ієрархічність файлової системи UNIX, що будується у вигляді дерева (рис. 2.1), не є складною. Існує навіть потужний засіб, який полягає у знятті обмеження на довжини імен файла і постфікса. Однак треба звернути увагу, що в ОС UNIX є 3 види файлових структур:

- звичайні дискові файли;
- каталоги;
- спеціальні файли.

2.1. Поняття про звичайні файли

Звичайні файли розміщуються на диску і містять інформацію, що в них записує користувач. Файлами також називаються готові до виконання програми, об'єктні модулі і т.і. ОС UNIX не накладає ніяких обмежень на внутрішню структуру інформації, збереженої у файлі.

Структурною інформацією керує користувач, а не система. Можуть існувати найрізноманітніші формати, відомі тільки програмам, які ці файли створюють або цими файлами користуються. За методологією побудова файлової системи – це дуже гнучкий механізм. З погляду ОС UNIX звичайний файл є безструктурним масивом байтів із прямим доступом.

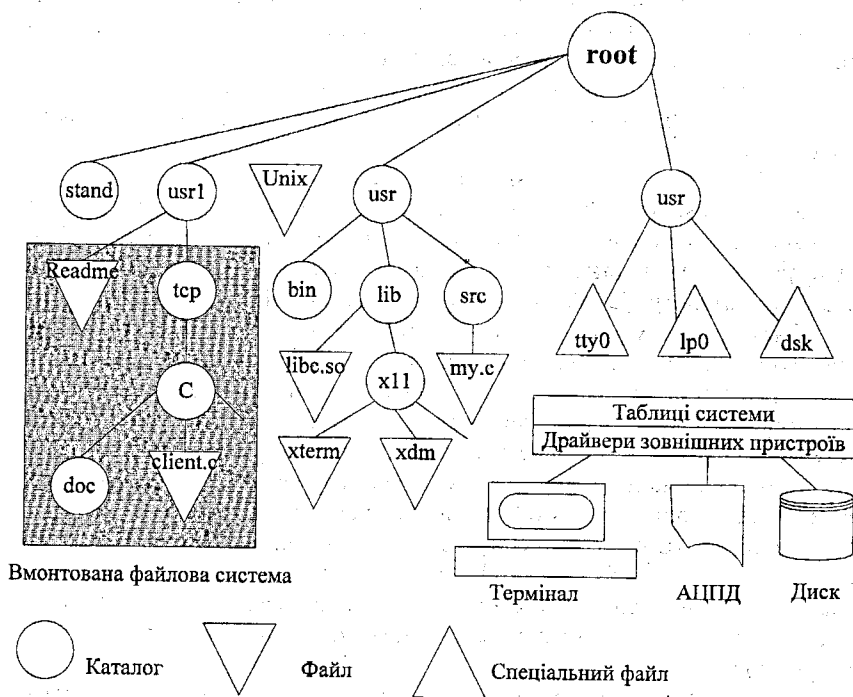


Рис. 2.1. Загальна деревоподібна структура файлової системи ОС Unix

Однак текстові файли в ОС UNIX прийнято формувати у вигляді рядків довільної довжини, відділених один від одного символом переведення рядка.

2.2. Потяття каталогів

Каталоги в загальному змісті – це прості папки, де зберігаються файли, згруповані за повною довільною ознакою. Наприклад, текстові документи, виконувані програми, бібліотеки і бібліотечні модулі, вихідні тексти програм і т.і. У свою чергу групи каталогів можуть утворити в логічному змісті том з головним кореневим каталогом, на який може бути змонтована будь-яка файлова система. Каталоги містять інформацію

про файли, у тому числі їхні імена, розміри, методи доступу, режими і типи. Оскільки каталоги містять важливу інформацію про файли, вони захищені механізмами ОС. Як і в звичайних файлах, для записування і читання інформації з каталогу вимагаються певні системні привілеї. В іншому (з погляду ОС), це такий же звичайний файл.

У системі застосовується універсальна угода про імена: повне ім'я складається з ланцюжка імен каталогів, через які проходить маршрут від кореня дерева (кореневого каталогу) до самого файла. Це – абсолютний тип файла. Наприклад, для вибірки файла client.c, що знаходиться в каталозі c, що знаходиться в каталозі tcp, що, у свою чергу, знаходиться в каталозі usr і, відповідно, у каталозі root (рис. 2.1), служить такий символічний рядок:

/usr/tcp/c/client.c

Якщо символічний рядок починається зі знака (/), то пошук починається з кореневого каталогу усієї файлової системи. Ім'я маршруту, що не починається зі знака (/), змушує ОС почати пошук з поточного каталогу. Такий тип файла носить назву відносного.

Система UNIX завжди пов'язує кожного користувача з визначенням каталогом. Коли користувач підключається до системи по команді login, він автоматично з'являється в каталозі (який інакше називають директорією) з ім'ям home. Цей каталог призначається адміністратором системи при реєстрації користувача. Якщо ім'я користувача приміром inter, то після команди login inter і введення пароля у відповідь на запит password, він входить до системи в каталог home/inter. Після цього можна по команді cd (Change Directory) змінити поточний каталог на який завгодно. За існуючої конвенції ім'я каталогу (..) завжди вказує на каталог, що знаходиться рівнем вище. Наприклад, якщо поточний каталог /home/inter, то виконання команди cd .. приведе до переходу в каталог /home.

Інша конвенція використовує крапку (.) як позначення поточного каталогу.

Внутрішня структура каталогу дуже проста: для кожного файла чи іншого каталогу нижнього рівня створюється один запис, організований в структуру такого змісту:

```
struct {  
    int inode;    /* індексний дескриптор */  
    char *name;   /* ім'я файла */  
}
```

Де inode містить номер індексного дескриптора, у якому зосереджена інформація про тип файла (каталог, звичайний файл чи спеціальний файл), про код його захисту, довжину, дату і час створення, а також розташування даних файла на диску. Існує по одному дескриптору на кожен файл, і саме з ними працює файлова система.

Звичайний користувач ОС UNIX не має, та й не повинний мати справи з індексними дескрипторами щонайменше доти, поки індексна

структура не порушена і не має потреби у відновленні. Але це вже справа системного адміністратора.

Для ОС UNIX введено таке поняття як дескриптор (описувач файлів). Це насамперед, є дуже важливим елементом ОС який дозволяє відокремити ім'я файла з яким працює користувач від інших специфічних даних з якими взаємодіє система. Такий підхід надзвичайно гнучкий і дозволяє маніпулювати зовнішнім представленням ієрархії файлів, не виконуючи зміщення самих файлів.

Зокрема, дуже просто той самий файл помістити в різні каталоги, створивши в них відповідні посилання на ім'я цього файла, але нікуди його фізично не переміщаючи. Файл може мати те саме ім'я в різних каталогах або імена-синоніми, але посилатися вони будуть на той самий індексний дескриптор, що є ключем для доступу до даних файла.

Кожне нове посилання з каталогів до індексного дескриптора відзначається в спеціальному полі. Це дозволяє файловій системі стежити за зайнятістю файла: як тільки лічильник посилань у результаті вилучення файла з каталогу стане рівним нулю, індексний дескриптор звільняється, а дисковий простір може бути використаний для запису інших файлів.

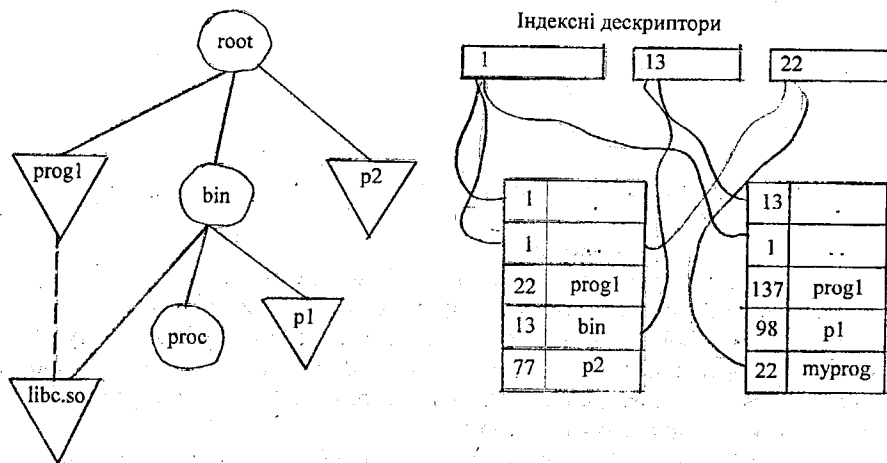


Рис. 2.2. Загальна схема фрагмента файлової системи зі змістом каталогів root і bin

Фрагмент файлової системи і зміст каталогів root і bin показані на рис.2.2. Каталог root є кореневим каталогом тому і має індексний дескриптор з номером 1. Файли з іменами які починаються на „..” чи „...” у каталозі root посилаються самі на себе, тому що для каталогу немає структури більш високого рівня. Важливо, що імена myprog і progl

посилаються на той самий індексний дескриптор, тобто на той самий файл.

Таким чином, файли і каталоги являють собою, логічну структуру файлової системи UNIX, яку можна представити у вигляді переверненого дерева, що має один корінь і безліч гілок. Кожна гілка такого дерева є каталогом (directory), а кожен листок на конкретній гілці – файлом (file) каталогу. Гілки стовбура називаються "батьківськими" каталогами каталогів і файлів, які вони містять усередині себе. Навпаки, каталоги і файли, що розташовані на основних гілках, називаються "дочірніми" каталогами (файлами) по відношенню до директорій, в яких вони містяться.

Слід зазначити, що логічна структура файлової системи UNIX завжди поза залежністю від кількості фізичних дискових пристроїв і кількості розділів на кожній з них, і має тільки один корінь (root). Ця директорія завжди позначається як (/) і називається кореневим каталогом файлової системи.

Перерахуємо найбільш важливі каталоги ОС UNIX на прикладі BSD та Solaris 2.5. До основних каталогів файлової системи можна віднести:

```
/
/var
/etc
/usr
/export
/opt
```

Розглянемо їх більш докладно.

2.2.1. Кореневий каталог /

Структуру файлової системи в UNIX можна представити у вигляді переверненого дерева, що має єдину початкову точку - так званий "корінь" файлової системи. Усі гілки дерева першого рівня "ростуть" саме від кореневої точки файлової системи. Важливо відзначити, що "корінь" файлової системи в UNIX завжди один, навіть якщо в системі є кілька фізичних дискових пристроїв.

Як правило, кореневий каталог файлової системи містить тільки інші підкаталоги. Для користувача root він є "домашнім каталогом" (home directory), і в ньому знаходяться стандартні файли користувача root (.login, .profile, .cshrc і т.і.).

Таким чином, при визначенні параметрів розміщення і розмірів кореневого каталогу існує тільки одна важлива вимога – у ньому повинно бути досить місця для збереження стандартних файлів користувача root. Всі інші підкаталоги кореневого каталогу можуть розміщатися в тому ж самому чи іншому розділах файлової системи.

Нижче приведено довільний зміст кореневого каталогу:

```
dev/      hosts      lib@      opt /      ndata
```

devices/	hsfsboot	log	proc/	vol/
etc/	iidw*	lost+found/	sbin/	etc.OLD/
iiup*	lpr.original/	tmp/	bin@	export/
iooffice/	mnt/	ufsboot	cdrom/	floppy/
kadb	net/	usr/	core	home/
kernel/	opt/			

2.2.2. Каталог /var

Каталог /var, як правило, містить інші підкаталоги для збереження різних конфігураційних файлів і файлів даних, що відносяться до різних системних сервісів. Дуже часто це "файли протоколів" (log file), у які записуються робочі повідомлення різних системних служб. Наприклад, /var/cron/log - файл повідомлень cron. В цьому каталозі можуть знаходитись тимчасові файли, у яких різні системні програми тимчасово зберігають свої дані. Наприклад, у каталозі /var/spool/lp розміщуються завдання на друкування, що чекають своєї черги. Важливо пам'ятати, що в цьому каталозі (/var/mail або /var/spool/mail) розміщуються файли "поштових скриньок" (mailbox) користувачів системи.

Таким чином, при початковому дисковому поділі для каталогу /var необхідно передбачити достатньо місця для збереження в /var/spool великих за розміром документів, що роздруковуються, великої кількості поштових скриньок користувачів і інше.

2.2.3. Каталог /etc

Каталог /etc є одним з найважливіших. Саме в ньому зосереджені основні конфігураційні файли і каталоги файлів ОС.

В цій директорії зберігаються дані про користувачів і їхні групи, інформація про хости, мережі. Тут також знаходяться завантажувальні файли (script files) які містять послідовність команд ОС, що повинні бути виконані в різних режимах завантаження системи.

Каталог /etc вимагає невеликого обсягу дискового простору (приблизно 8-15 Мбайт) і, оскільки вміст директорії є надзвичайно важливим для правильної роботи системи, рекомендується регулярно зберігати вміст цього каталогу. Вважається правильним обов'язково робити копію перед тим, як що-небудь необхідно змінити в конфігурації системи. Це ж відноситься і до каталогу /var.

2.2.4. Каталог /usr

Каталог /usr в основному призначений для збереження виконуваних файлів системи, бібліотек, системних додатків і т.і. В багатьох випадках ці файли входять до складу ОС і необхідні для всіх користувачів системи. Набір файлів, що містяться в цьому каталозі залежить від конкретної реалізації ОС. Наприклад, у стандарті BSD прийнято зберігати в цьому каталозі та у підкаталогах також і файли, що не входять у стандартний склад ОС. Ці файли можуть бути додані

пізніше адміністратором і доступні для всіх користувачів системи. До них відносяться виконувані модулі текстових редакторів, мов програмування, додаткових системних утиліт з керування файловою системою або програм, які виконують моніторинг системи.

Дисковий простір, який необхідний для цього каталогу, залежить від конкретної ОС і набору системних програмних продуктів. Для прикладу, в ОС Solaris це, як правило, близько 180 Мбайт (повний набір системних продуктів, включаючи графічні оболонки), а в ОС Linux – близько 120 Мбайт.

2.2.5. Каталог /home (/export/home)

Каталог /home призначений для збереження підкаталогів, що представляють собою домашні директорії користувачів системи. Каталог з ім'ям \$HOME присутній у будь-якій реалізації ОС UNIX. Але конкретний механізм роботи з цим каталогом і його фізичним розташуванням можуть відрізнятися навіть між однаковими реалізаціями ОС на різних комп'ютерах. Великою мірою це залежить саме від адміністратора системи. Саме він визначає спосіб розміщення домашніх каталогів користувачів.

Можна виділити два способи розміщення домашніх каталогів (за умови, що є комп'ютерна мережа).

По-перше, домашні каталоги користувачів можуть розміщатися локально, на тих машинах, де ці користувачі працюють. Якщо користувач може працювати на декількох машинах, то на кожній з них він має свій домашній каталог.

По-друге, домашні каталоги всіх користувачів можуть розміщатися на якій-небудь одній машині в мережі (назвемо її `home_server`) і тільки на ній, у спеціальному каталозі (назвемо його `/export/home`). У випадку, коли користувачу необхідний доступ до його домашнього каталогу, наприклад, якщо існує входження у систему, то його директорія динамічно монтується (NIS, NIS+ чи NDS) з `home_server` (з каталогу `/export/home`) на той комп'ютер, де виконується `login` (на каталог `/home`). Таким чином, кожен користувач має єдиний домашній каталог поза залежністю від структури мережі в якій він може працювати. В цьому випадку, на кожній з машин він має однакове ім'я. Важливо, що другий спосіб є кращий тому що, не "прив'язує" користувача до конкретної машини, а автоматично створює йому ідентичні умови для роботи на будь-якому доступному в мережі комп'ютері. Більш того, усі дані користувача локалізовані в одному місці, що звільняє його від необхідності переносити їх з однієї машини на іншу. Мабуть, єдиним недоліком другого способу є те, що дані користувача повинні передаватися по мережі. Це може стати проблемою в тому випадку, якщо використовується низькошвидкісна чи перевантажена мережа, а користувач у процесі роботи застосовує прикладні додатки, що інтенсивно експлуатують тимчасові файли в його домашньому каталозі.

2.2.6. Каталог /opt (/opt1, /opt2, ... /optn)

Наявність або відсутність каталогу /opt залежить від конкретної реалізації ОС UNIX. Наприклад, у ОС Solaris він, як правило, є, а в Linux, BSD або SCO – його немає. Якщо ця директорія присутня, то містить додаткове програмне забезпечення, яке доступне для всіх користувачів системи. Наприклад, це можуть бути засоби розробки додатків, видавничі системи, СУБД і т.і.

Розмір дискового простору, необхідного для каталогу /opt, визначається кількістю програмних засобів, які передбачається інсталивати в даній системі.

Найчастіше, якщо в системі встановлюється додатковий дисковий пристрій, він монтується на каталог /opt (/opt1, /opt2, ..., /optn).

Filesystem	kbytes	used	avail	capacity	Mounted on
/dev/dsk/c0t3d0s0	901526	574269	237107	71%	/
/dev/dsk/c0t1d0s2	1970068	1778926	92642	95%	/opt

2.3. Сучасні механізми роботи з оперативною пам'яттю

2.3.1. Призначення спеціального розділу SWAP

Файл swar являє собою спеціальний розділ файлової системи UNIX. В основному, він призначений для розширення віртуальної пам'яті, доступної додаткам у системі та у тих випадках, коли активному додатку потрібно більше оперативної пам'яті, ніж у даний момент є в системі (чи взагалі є в системі).

Саме через файл swar здійснюється свопінг та пейджинг. Якщо swar - розділ дискового пристрою, то він, як правило, монтується на будь-який каталог. У ОС Solaris для цієї мети за замовчуванням (не зовсім вдало за методами системного адміністрування) використовується директорія /tmp :

Filesystem	kbytes	used	avail	capacity	Mounted on
dev/dsk/c0t1d0s0	30991	13379	14522	48%	/
dev/dsk/c0t1d0sG	222439	175159	25040	87%	/usr
dev/dsk/c0t0d0s6	38493	28173	6480	81%	/var
swap	50688	8	50680	0%	/tmp
dev/dsk/c0t1d0s3	38207	45	34342	0%	/export
dev/dsk/c0t1d0s4	28831	9	25942	0%	/export/swap
dev/dsk/c0t0d0s5	574342	486041	30871	94%	/opt

2.3.2. Рекомендації по керуванню розділом swar

Жорстких вимог до розміщення і розміру swar немає, є тільки такі рекомендації.

Початковий розмір swar у системі (розмір, що вказується при початковій інсталяції ОС) краще встановлювати рівним двом-трьом

розмірам фізичної оперативної пам'яті, але це число залежить від розміру фізичної оперативної пам'яті в системі і від додатків, що будуть виконуватися в системі, наприклад, якщо фізичний розмір оперативної пам'яті в системі складає 4096 Мбайт, то, можливо, swap не буде потрібний зовсім, а якщо 16 Мбайт, то напевно буде потрібний swap розміром не менше 64 Мбайт.

Розміщувати swap (розділи і файли) найкраще на найшвидшому дисковому пристрої в системі.

Swap може складатися з декількох частин, наприклад з одного розділу на диску та одного чи декількох додаткових swap-файлів, що можуть динамічно підключатися або відключатися від системи. Варто прагнути мінімізувати кількість постійно використовуваних частин, що складають swap. В ідеальному випадку – найкраще мати один розділ на дисковому пристрої, відведеному під swap.

2.3.3. Команди керування swap розділом

До основних команд керування swap можна віднести команди swap і mkfile. Команда swap має синтаксис:

```
/usr/sbin/swap -a swapname [ swaplow ] [ swaplen ]  
/usr/sbin/swap -d swapname [ swaplow ]  
/usr/sbin/swap -l  
/usr/sbin/swap -s
```

де параметр

- a swapname дозволяє додати зазначену swap-область. Цей ключ може бути використаний тільки суперкористувачем (root), swapname - це ім'я swap-файла, наприклад: /dev/dsk/c0t0d0s1 або звичайний файл;
- d swapname дозволяє вилучити зазначену swap-область. Цей ключ може бути використаний тільки суперкористувачем (root), swapname - це ім'я swap-файла, наприклад: /dev/dsk/c0t0d0s1 чи звичайний файл;
- l дозволяє показати статус усіх swap-областей;
- s дозволяє показати сумарну інформацію про загальне використання swap-простору.

В загальному випадку команда swap дозволяє додавати, вилучати та стежити за станом області swap.

Синтаксис команди mkfile в ОС UNIX може складати:

```
mkfile [ -nv ] size[k|b|m] filename ...
```

де параметр:

- n створює порожній файл. Розмір файла позначається, але дискові блоки не розміщуються доти, поки у файл не записуються дані.
- v виводить повідомлення про ім'я і розмір створеного файла.

В загальному випадку mkfile створює один чи більше файлів, придатних для використання в якості NFS-монтовних чи swap-областей у якості локальних swap-областей. Для цього встановлюється біт і за замовчуванням файл заповнюється нулями. Розмір файла за замовчуванням задається в байтах, але може бути зазначений у

кілобайтах, блоках або мегабайтах за допомогою вказівки відповідно: k, b чи m суфіксом.

2.3.4. Додатковий моніторинг використання swap

Додатковий моніторинг swap області в ОС UNIX може бути виконаний за допомогою команди `vmstat`. Ця команда відображає статистику використання віртуальної пам'яті і має синтаксис:

```
vmstat [ -cis ] [ disks ] [ interval [ count ] ]
```

де параметр

`cis` – параметри системних процесів;

`disks` – диск або розділ в якому знаходиться swap;

`interval [ count ]` – інтервал або проміжок часу за який необхідно підраховувати статистику.

В загальному випадку, команда `vmstat` відображає статистику про процеси, роботу віртуальної пам'яті, активності жорсткого диск, CPU і інше.

Наприклад, якщо користувач виконає команду:

```
vmstat -cis /dev/dsk/c0t0d0s0
```

він може побачити на екрані монітора:

```
procs memory page      diskfaults  cpu
r  b  w  swap free re mf  pi  po  sr  f0  s1  s3  in  sy  cs  us  sy  id
0  0  0  4352  1528   1  12  13  0  4  0  2  0  102 942 262 8  4
88   0  0  0  48700 1424   0  67  0  0  0  0  0  0  76 598 171
1  12 87   0  0  0  48692 1420   0  1  0  0  0  0  0  0  62
539 142 0  0  100   0  0  0  48692 1420   0  0  0  0  0  0  0
0  61 691 148 0  0  100   0  0  0  48692 1420   0  0  0  0  0
0  0  0  62 640 145 0  0  100
0  0  0  48692 1420   0  0  0  0  0  0  0  0  63 635 151 0  0
100
0  0  0  48692 1420   0  0  0  0  0  0  0  0  61 571 135 0
10 90
0  0  0  48692 1420   0  0  0  0  0  0  0  0  63 626 146 0  0
100
0  0  0  48692 1420   0  0  0  0  0  0  0  0  62 580 142 0  0
100 0  0  0  48692 1420   0  0  0  0  0  0  0  0  62 624 154
0  0  100
0  0  0  48692 1420   0  0  0  0  0  0  0  0  62 618 140 0  0
100 0  0  0  48692 1420   0  0  0  0  0  0  0  0  61 571 147
0  6  94
```

2.4. Поняття про спеціальні файли

Однією з потужних особливостей ОС UNIX є система введення/виведення, де всі зовнішні пристрої розглядаються як файли. Вони передбачають виконання над собою звичайних файлових операцій.

Оскільки в ОС UNIX існують і драйвери пристроїв, то інтерфейс користувача з ними розроблений по звичайному типу у вигляді спеціального файлу. Так, кожному підключеному пристрою, наприклад, терміналу, диску, пристрою друкування, стримеру і т.і. відповідає як мінімум один спеціальний файл. Велика кількість їх зберігається в каталозі /dev . Якщо виконати команду:

```
$ cd /dev
```

```
$ ls -l
```

буде виведена велика кількість спеціальних файлів. Деякі з них позначають:

ім'я спеціальних файлів	Позначка
console	пульт керування системою
dsk	розділ диска
fd0	флопі-диск 1
fd1	флопі-диск 2
mem	пам'ять
lp	принтер
lp0	паралельний порт 0
lp1	паралельний порт 1
rmt	магнітна стрічка (стрімер)
scsi	SCSI контролер
swap	своп розділ
syscon	ім'я для системної консолі
systty	ще одне ім'я для системної консолі
term	термінальні пристрої
tty00	серійний порт 0 (COM1)
tty00h	
tty00s	
...	

Коли програма виконує запис в такий файл, наприклад у /dev/rmt то система перехоплює їх і направляє на стрімер. При читанні даних з цього файлу в дійсності вони передаються з пристрою, до якого підключена магнітна стрічка. Програма не повинна враховувати особливості роботи пристрою введення/виведення. Для цієї мети і служать спеціальні файли, що виконують функції інтерфейсу між компонентами ядра ОС (драйвери) і прикладними програмами загального призначення.

Система виявляє відмінність звичайного файлу від спеціального тільки після того, як буде проаналізований відповідний індексний дескриптор, на який посилається запис у каталозі.

Індексний дескриптор спеціального файлу містить інформацію про клас пристрою, його тип і номер.

2.4.1. Клас пристрою спеціальних файлів

Деякі пристрої введення/виведення працюють посимвольно, інші - поблочно. Типовим прикладом пристрою з посимвольним обміном може

служити термінал комп'ютера. В цьому випадку комп'ютер здійснює обмін даними з терміналом у побайтовому режимі. Спеціальні файли, що забезпечують зв'язок із пристроями такого типу, називають байт-орієнтованими.

Для блокових пристроїв характерний обмін великими блоками інформації для прискорення обміну і виконання цього процесу більш ефективно. Так працюють усі дискові пристрої, що називаються блоковими, а спеціальні файли, що обслуговують їх – блок-орієнтованими.

Спеціальні файли не містять будь-якої символічної інформації, тому в лістингу каталогу їх довжина не вказується.

2.4.2. Тип і номер спеціальних файлів

Для спеціальних файлів у полі довжини містяться головний і додатковий номери відповідного пристрою. Перший з них визначає тип пристрою, другий – ідентифікує його серед однотипних. Наприклад, ОС UNIX може одночасно обслуговувати кілька десятків і сотень терміналів. Кожний з них повинен мати свій власний спеціальний файл. Тому наявність головного і додаткового номерів дозволяє встановити необхідну відповідність між пристроєм і таким файлом.

Таким чином, спеціальні файли є засобом уніфікації введення/виведення в ОС UNIX. У своїй сукупності вони утворюють іншу важливу особливість файлової системи UNIX – її фізичну структуру. Фізична структура файлової системи залежить від кількості наявних дискових пристроїв і способу їх поділу на розділи. Будь-який дисковий пристрій має як мінімум один розділ. Максимальна кількість розділів на одному пристрої визначається конкретною реалізацією UNIX. Фізично розділ являє собою набір визначених міток на дисковому пристрої і запис у таблиці поділу диска. Як правило, у перших циліндрах диска знаходиться інформація про координати на диску і розміри розділу.

2.4.3 Зв'язок фізичної і логічної структури дискового простору

Кожний розділ містить як мінімум один каталог. Цей каталог можна розглядати як кореневий каталог для даного розділу дискового пристрою. Як і будь-який інший цей каталог може містити підкаталоги. Зв'язок між розділом і таким каталогом встановлюється в процесі монтування (mount) файлової системи. При цьому розділ дискового пристрою обов'язково монтується на каталог. Цей зв'язок ілюструється таким прикладом для ОС BSD або Solaris, якщо виконати команду:

```
$ df -k
```

Filesystem	kbytes	used	avail	capacity	Mounted on
dev/dsk/c0tld0s0	30991	13379	14522	48%	/
dev/dsk/c0tld0s6	222439	175159	25040	87%	/usr
dev/dsk/c0t0d0s6	38493	28173	6480	81%	/var

swap	50688	8	50680	0%	/tmp
dev/dsk/c0tld0s3	38207	45	34342	0%	/export
dev/dsk/c0tld0s4	28831	9	25942	0%	/export/swap
dev/dsk/c0t0d0s5	574342	486041	30871	94%	/opt

У даному випадку найбільший інтерес представляють перший та останій стовпчики: "Filesystem" та "Mounted on". "Filesystem" – це розділ дискового пристрою, а "Mounted on" – каталог файлової системи, на який змонтований розділ. У даному прикладі є два дискових пристрої: `dev/dsk/c0t0d0[s*]` і `/dev/dsk/c0tld0[s*]`. Пристрій `/dev/dsk/c0t0d0[s*]` розділений на два розділи `*[s5,s6]`, а пристрій `/dev/dsk/c0tld0[s*]` розділено на чотири розділи `*[s0,s3,s4,s6]`, кожний з яких змонтований на свій каталог (відповідно `/opt`, `/var`, `/i` `/export`, `/export/swap`, `/usr`).

Про дисковий поділ було сказано в цьому розділі. Однак, треба зазначити, що поділ дискового простору спочатку виконується при інсталяції ОС UNIX і є логічним продовженням утиліти `FDISK`. На відміну від `MS DOS`, на кожному розділі ОС UNIX може бути створена додаткова кількість дискових поділів. За термінологією це позначається як `slice`. Ці участки дискового простору потім можуть використовуватися для створення файлових систем і для іншої мети. Додатковий дисковий поділ може бути виконаний за допомогою спеціальних утиліт, наприклад, `divvy` в `SCO UNIX`.

З викладеного матеріалу ясно, що каталог не може мати розмір більший, ніж розмір розділа, на який він змонтований. Вибір способу поділу дискового пристрою значною мірою залежить від задач, які передбачається вирішувати на конкретній системі.

Деякі роки назад вважалося, що поділ дискового пристрою на кілька розділів у деяких випадках може дати перевагу у продуктивності системи. Але сьогодні, з появою нових дискових пристроїв, удосконалюванням їхніх контролерів і реалізацією потужних файлових систем, таке твердження вже не видається обґрунтованим. Першими від цього відмовились розроблювачі `HP-UX`, у якій завжди був тільки один гігантський розділ. Це було пов'язано з тим, що поділ одного дискового пристрою на кілька розділів не давав значного виграшу в продуктивності, зате створювались додаткові труднощі у використанні дискового простору. Наприклад, коли є два розділи, у кожному з яких залишилося по 100 Мбайт вільного простору, а користувачу необхідно мати 150 Мбайт одним сегментом. Крім того, зайве збільшення кількості розділів приводить до росту неефективної витрати дискового простору на системні потреби.

2.5. Створення файлової системи мережових ОС

Нову файлову систему можна створити за допомогою команди `mkfs` (`make file system`). Наприклад, вираз `# /sbin/mkfs /dev/dsk/flt 512`

означає: створити файлову систему на флопі-диску в розміром у 512 блоків.

На одному диску можна створити кілька файлових систем. Деякі системи UNIX використовують по одній файловій системі на диску, а інші по декілька. Причини цих різних підходів скоріше філософські, ніж практичні.

Після того як mkfs почне працювати, вона дає можливість відмовитися, перш ніж продовжувати. Після того як виведеться повідомлення

```
Mkfs: /dev/dsk/flt?
```

```
(Del if wrong)
```

команда очікує 10 с. перш ніж почати створення файлової системи. Якщо користувач вирішив відмовитися, він повинен встигнути натиснути клавішу DEL. Якщо команда продовжить роботу, вона виведе інформацію про файлову систему, включаючи кількість логічних блоків, кількість і-вузлів і кількість блоків файлової системи, у які можна записувати дані. Наприклад:

```
Mkfs: /dev/dsk/flt?
```

```
(Del if wrong)
```

```
bytes per logical block = 1024
```

```
total logical blocks = 256
```

```
total inodes = 64
```

```
gap (physical blocks) = 7
```

```
cylinder size (physical blocks) = 400
```

```
mkfs: Available blocks = 248
```

За бажанням користувача можна задати другий аргумент. Тому, в загальному вигляді перше число означає розмір у блоках, а друге – кількість і-вузлів (тобто максимальне число файлів, що можуть бути збережені у файловій системі). За замовчуванням число і-вузлів дорівнює числу блоків, діленому на 4. Максимальне число і-вузлів в одній файловій системі 65 000. Якщо з деяких причин потрібно більш 65000 і-вузлів на диску, необхідно створити дві або більше файлові системи на цьому диску.

2.6. Монтовність і демонтовність файлових систем

Одна з потужних характеристик файлових систем UNIX – це можливість монтування. Це означає, що файлова система може бути прикріплена до загального дерева каталогів у будь-якій його точці. Наприклад, каталог / - це кореневий (root) каталог системи. Крім того, він є основою файлової системи, що завжди вмонтована. Каталог /usr1 знаходиться в каталозі /, але в даному випадку є окремою файловою системою від кореневої, тому що всі файли в ньому знаходяться на окремій частині диска або взагалі на окремому диску. Файлова система

/usr1 - монтовна система, корінь якої знаходиться у точці /usr1 та існує в загальній ієрархії (рис. 2.3 і 2.4).

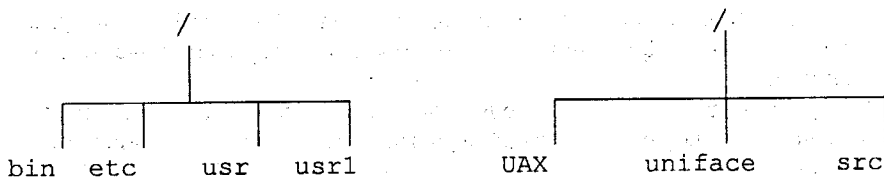


Рис. 2.3 Дві файлові системи перед монтуванням /dev/dsk/osi

Для монтування файлової системи використовується команда /sbin/mount. Ця команда дозволяє об'єднати дану файлову систему з існуючою в структурі каталогу:

команда	пояснення
#/sbin/mount /dev/dsk/osi /usr1	монтування жорсткого диска на каталог /usr1
#/shin/mount /dev/dsk/flt /b	монтування флопі-диска /dev/dsk/flt на каталог /b

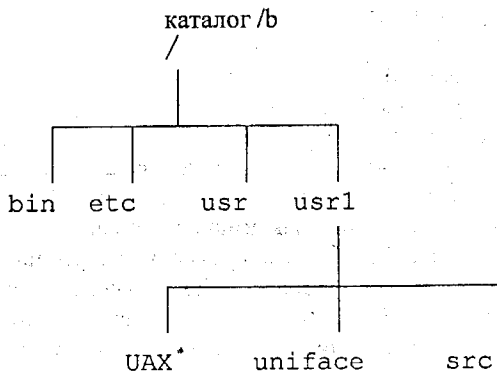


Рис. 2.4 Файлова система після монтування диска /dev/dsk/osi на каталог /usr1

У випадку якщо користувач хоче монтувати файлову систему, щоб вона була доступна тільки для читання, можна додати опцію -r до команди /sbin/mount.

Коли монтується файлова система необхідно бути впевненим, що каталог, до якого вона прикріплюється, у даний момент порожній. Це необхідно тому, що вміст його стане недоступним, поки файлова система буде монтуватися.

Щоб одержати інформацію про файлові системи, що змонтовані на вашій системі UNIX, можна використовувати команду `/sbin/mount` без аргументів:

```
# /sbin/mount
/ on dev/root read/write/setuid on Wed Dec 14 07:54:21 1994
/usr1 on /dev/dsk/cot1dos1 read/write
/usr2 on /dev/dsk/cot1dos2 read/write/log/setuid/mincache = dosesync on
/b on /dev/dsk/fit read/write/setuid on
Wed Dec 14 16:23:52 1994
```

Команда `/sbin/mount` виводить каталог, на якому була змонтована файлова система. Цей приклад показує пристрій `/dev`, на якому знаходиться змонтована файлова система, дата монтування та атрибути можливих операцій над прикріпленими дисками.

Процеси можуть звертатися до файлів у змонтованій файловій системі та ігнорувати той факт, що каталог може від'єднатися у будь-який час. Системна функція `link` контролює цілісність файлової системи. Тому, у багатьох версіях ОС UNIX не дозволяються зв'язки між файлами, що належать різним файловим системам.

Ядро підтримує таблицю монтування з записами про кожну змонтовану файлову систему. У записі таблиці монтування існують:

- номер пристрою, що ідентифікує змонтовану файлову систему (логічний номер файлової системи);
- показчик на буфер, де знаходиться суперблок файлової системи;
- показчик на кореневий індекс змонтованої файлової системи;
- показчик на індекс каталогу, що став точкою монтування.

Зв'язок індексу точки монтування з кореневим індексом змонтованої файлової системи, що виникла в результаті виконання системної функції `mount`, дає ядру можливість пересування по ієрархії файлових систем без одержання від користувачів додаткової інформації.

Для демонтування файлової системи використовується команда `/sbin/umount`. Ця команда має зворотну дію порівняно з командою `mount`. Вона звільняє файлову систему і вилучає її зі структури каталогів, таким чином що всі її власні файли і каталоги стають недоступні:

```
# /sbin/umount /b
```

або

```
# /sbin/umount /dev/dsk/0s2
```

Необхідно зауважити, що коренева файлова система не може бути демонтована. Крім того, команда `umount` не буде виконана, якщо хто-небудь використовує файл або каталог із тієї файлової системи, яку намагаються демонтувати. Це може бути навіть просте перебування деякого користувача в одному з каталогів файлової системи, що демонтується.

Команди mount і umount також відносяться до числа не багатьох команд в ОС UNIX, де користувач зустрічається з аббревіатурою фізичних дискових пристроїв.

2.6.1 Дискові пристрої в ОС UNIX

Дискові пристрої в ОС UNIX мають складні і часом незрозумілі користувачам, які працювали з ОС MS DOS або Windows, позначення. Але в ОС UNIX користувач ніколи не зустрічається з проблемою точної вказівки фізичного пристрою, на якому розташовується інформація. В багатьох випадках цей процес значно спрощується тим, що користувач має справу тільки з ім'ям каталогу, у якому знаходиться потрібна йому інформація.

Усі файлові системи монтуються один раз, як правило, при завантаженні системи. Більш того, на деякі каталоги можуть бути змонтовані файлові системи з віддалених машин комп'ютерної мережі.

Для фізичних пристроїв існують директорії dsk і ulsk, що містять спеціальні файли. Імена файлів у цих директоріях однакові і єдина різниця між ними та, що директорія rdsk містить дискові пристрої зі спеціальним доступом (raw). Це дозволяє використовувати деякі пристрої системи для більш швидкого доступу до диска. Одна типова директорія dsk може містити такі пристрої:

```
$ ls /dev/dsk
```

0s0	1s0	c0t0d0s0	c0t1d0s0	f0	f05q	f13dt	f1d8d
0s1	1s1	c0t0d0s1	c0t1d0s1	f03d	f05qt	f13h	f1d8dt
0s2	1s2	c0t0d0s2	c0t1d0s2	f03dt	f0d8d	f13ht	f1d8t
0s3	1s3	c0t0d0s3	c0t1d0s3	f03h	f0d8dt	f15d16	f1d9d
0s4	1s4	c0t0d0s4	c0t1d0s4	f03ht	f0d8t	f15d16t	f1d9dt
0s5	1s5	c0t0d0s5	c0t1d0s5	f05d16	f0d9d	f15d4	f1d9t
0s6	1s6	c0t0d0s6	c0t1d0s6	f05d16t	f0d9dt	f15d4t	f1q15d
0s7	1s7	c0t0d0s7	c0t1d0s7	f05d4	f0d9t	f15d8	f1q15dt
0s8	1s8	c0t0d0s8	c0t1d0s8	f05d4t	f0q15d	f15d8t	f1q18dt
0s9	1s9	c0t0d0s9	c0t1d0s9	f05d8	f0q15dt	f15d8u	f1q9dt
0sa	1sa	c0t0d0sa	c0t1d0sa	f05d8t	f0q18d	f15d9	flt
0sb	1sb	c0t0d0sb	c0t1d0sb	f05d8u	f0q18dt	f15d9t	
0sc	1sc	c0t0d0sc	c0t1d0sc	f05d9	f0q9dt	f15h	
0sd	1sd	c0t0d0sd	c0t1d0sd	f05d9t	f0t	f15ht	
0se	1se	c0t0d0se	c0t1d0se	f05h	f1	f15q	
0sf	1sf	c0t0d0sf	c0t1d0sf	f05ht	f13d	f15qt	

У системі UNIX дискові пристрої логічно розділені на секції подібно тому як у MS DOS можуть існувати логічні диск у Partition Table. В цьому прикладі ми бачимо, що файли 0s0, 0s1, 0s2 і т.д. відповідають секціям один, два, три і т.д. відносно диска з номером 0. Інші файли: 1s0, 1s1, 1s2 і т.д. відповідають секціям один, два, три і т.д. відносно диска з номером 1.

Якщо система має більше дисків, то секції будуть пронумеровані ns0 ns1 і т.д. для кожного диска з номером n. Системи з великою кількістю дискових пристроїв дотримуються такої системи нумерації:
c controller d disk s section

де

controller - номер контролера диска;

disk - номер диска;

section - номер секції диска.

Так, 0s0 звичайно еквівалентно c0t0d0s0, 0s1 – c0t0d0s1, і трисимвольні імена секцій - це просте скорочення для дискового контролера з номером 0.

Файли, у яких імена починаються з f, визначають різні види флоппі-дискових пристроїв. Каталог rmt містить файли на пристроях типу магнітної стрічки:

```
$ ls /dev/rmt
```

```
cos0 з ls0 c3s0 ntape ntape1 tape tape1 *
```

Файли c0s0, ls0, c2s0 і c3s0 визначають чотирикасетні стрічкові запам'ятовувальні пристрої. Файли tape і tape1 визначають магнітні запам'ятовувальні пристрої із двома бобінами. Файли, чиї імена починаю з n, відносяться до тих же пристроїв. В цьому випадку, стрічка не перемотується після використання, у той час як використання інших файлів змушує стрічку перемотуватися після закінчення роботи програми.

В інших системах UNIX ці файли мають інші назви, однак важливо те, що усі вони завжди знаходяться в каталозі /dev . Словник, що обов'язково приходить із системою, містить докладний опис пристроїв і пов'язаних з ними файлів.

Контрольні питання

1. Назвіть файлові системи, які може підтримувати ОС Unix.
2. Загальна характеристика функцій файлової системи ОС Unix.
3. Особливості та відмінності файлової системи ОС Unix від ОС MSDOS.
4. Загальна характеристика звичайних файлів.
5. Загальна характеристика каталогів. Поняття про відносне, абсолютне та поточне імена каталогів.
6. Поняття та особливості у використанні індексного дескриптора. Логічна структура файлової системи ОС Unix.
7. Загальна характеристика основних каталогів ОС Unix: /, /home, /etc, /var, /usr .
8. Призначення файла swap, основні властивості. Методи керування файлом swap.
9. Характеристика та синтаксис команд керування файлом swap: swap, mkfile, vmswap.

10. Загальна характеристика спеціальних файлів та їх призначення. Приклади спеціальних файлів. Зв'язок фізичної і логічної структур дискового простору.

11. Методи створення файлової системи. Особливості використання команд. Можливість існування декількох ОС на жорсткому диску.

12. Монтівність та демонтівність файлової системи ОС Unix. Призначення, особливості використання, приклади.

Контрольні вправи

1. Виконайте перехід в інший каталог, використовуючи відносні, абсолютні та поточні імена каталогів.

2. Побудуйте графічну структуру основних каталогів ОС Unix: /, /home, /etc, /var, /usr .

3. Визначте розмір файла swar та основних розділів ОС Unix.

4. За допомогою команд керування файлом swar:

- створіть файл swar;

- отримайте статистику про процеси, віртуальну пам'ять, загальні показники жорсткого диска, активність CPU;

- отримуйте інформацію про статус всіх swar-областей, загальну інформацію про сумарне використання swar-простору;

- вилучте swar-простір.

5. За допомогою команд ОС Unix знайдіть каталог, визначте та дайте стислу характеристику знайденим спеціальним файлам.

6. За допомогою команд ОС Unix створіть файлову систему на флопі-диску розміром в 512 блоків.

7. Виконайте монтування файлової системи флопі-диска на жорсткий диск. Визначте розміри вільного дискового простору флопі та жорсткого дисків.

2.7. Внутрішня структура мережевих операційних систем

2.7.1. Розподіл дискового простору

Розподіл дискового простору має велике значення для знайомства з внутрішньою структурою мережевих ОС. Розглянемо більш детально структуру файлової системи на диску.

Перший блок файлової системи – це блок початкового завантаження. Якщо носій передбачає початкове завантаження, то цей блок містить коротку програму-завантажник. За допомогою засобів завантажника ядро ОС розміщується в пам'яті та отримує керування.

Наприклад, для машин серії 300/400 Hewlett Packard область завантаження займає 8 Кбайт і містить таку інформацію:

Volume

Header 256 bytes

Unused

256 bytes

Розділ Volume directory information містить 3 імені: SYSHPUX, SYSBCKUP і SYSDEBUG. SYSHPUX відповідає файлу /hp-ux який є ядром системи. SYSBCKUP відповідає файлу /sysbckup, що використовується при збереженні системи. SYSDEBUG відповідає файлу /sysdebug, що використовується при написанні драйверів.

Останні 7,25 Кбайт області завантаження містять так званий вторинний завантажник (secondary loader). Boot ROM завантажує і передає керування вторинному завантажнику, що за бажанням користувача може завантажити будь-який з трьох перерахованих файлів, що містять ядро ОС.

Другий блок також має довжину 8 Кбайт. Він займає суперблок, що містить заголовок файлової системи та основну інформацію про неї, що включає: розмір, кількість вільних блоків, індексних дескрипторів та іншу інформацію. При монтуванні файлової системи по команді mount у таблиці змонтованих систем, розташованій в ядрі, формується відповідний елемент, а суперблок записується в один з великих внутрішніх буферів ядра. Потім система працює із суперблоком для пошуку і відновлення списку вільних і зайнятих блоків на диску.

Дисковий простір, відведений під файлову систему, займає блоки, об'єднані в так званий і-список (i-nodes) або список індексних дескрипторів.

2.7.2. Індексні дескриптори

Логічне представлення файла відрізняється від його фізичного представлення. Логічне представлення – це те, що можна бачити, якщо роздруковувати файл. В цьому випадку користувач одержує потік символів, що представляють вміст файла. Фізичне представлення - це реальна організація файла на диску. Файл, розмір якого перевищує один блок, використовує складну структуру диска. Коли користувач здійснює доступ до файла, система UNIX знаходить окремі блоки в правильному порядку та організовує логічне представлення інформації у файлі. Це і є відмінністю роботи з дисковим простором ОС UNIX по відношенню до інших систем. При роботі з деякими ОС необхідно вказати номери початкового і кінцевого блоків файла. Але такі файлові системи незручні, оскільки в них збільшення розмірів файлів обмежена сусідніми наборами. У файлових системах з неперервними файлами виникають проміжки між файлами і необхідно регулярно проводити ущільнення або компресію диска. У файловій системі UNIX не існує цих недоліків.

Зрозуміло, що при такій організації файлів у ОС UNIX повинен існувати список, який би показував, як конвертувати фізичне

представлення файлу в логічне. Це і є список індексних дескрипторів або і-вузлів (i-nodes).

Кожен файл визначається індексним дескриптором, що містить всю необхідну інформацію про даний файл: режим, тип файлу, його довжина в байтах, ідентифікатори власника і групи, дата останньої модифікації. Довжина індексного дескриптора складає 64 байта, його структуру можна знайти в файлі /usr/include/sys/inode.h. Розглянемо більш детально цей файл:

```
#ifndef FS_XX_INODE_H /* wrapper symbol for kernel use */
#define FS_XX_INODE_H /* subject to change without notice */
#define "(@(#)uts-x86:fs/xx/inode.h 1.2"
#define $Header: $"
#define FSPTR 1
/*
 * The I node is the focus of all file activity in unix.
 * There is a unique inode allocated for each active file,
 * each current directory, each mounted-on file, text file,
 * and the root. An inode is 'named' by its dev/inumber
 * pair. (iget/iget.c) Data, from mode on, is read in from
 * permanent inode on volume.
 */2
struct iisem { /* XENIX semaphore */
    short i_scount; /* current semaphore count */
    short i_eflag; /* err flag */
    struct file *i_headw; /* first waiter */
    struct file *i_tailw; /* last waiter */
};
struct iisd { /* XENIX shared data */
    union {
        struct region *i_region; /* Pointer to the shared region */
        struct iisd *i_chain; /* next available shared */
    } i_iun; /* data structure */
    unsigned i_len; /* limit of segment (seg size - 1) */
    short i_snum; /* serial # for sdgetv, sdwaitv */
    short i_flags; /* LOCKED, etc. */
};
typedef struct inode
{
    struct inode *i_forw; /* inode hash chain */
    struct mode *i_back; /* " */
    struct inode *av_forw; /* freelist chain */
    struct inode *av_back; /* " */
    int *i_fsptr; /* "typeless" pointer to fs dependent */
    long i_number; /* i number, 1-to-1 with dev address */
}
```

```

ushort i_ftype; /* file type = IFDIR, IFREG, etc. */
short i_fstyp; /* File system type */
off_t i_size; /* size of file */
ushort i_uid; /* owner */
ushort i_gid; /* group of owner */
ushort i_flag;
ushort i_fill
cnt_t i_count; /* reference count */
short i_nlink; /* directory entries */
dev_t i_rdev; /* Raw device number */
dev_t i_dev; /* device where inode resides */
struct mount *i_mntdev; /* ptr to mount dev inode */
/* resides on */

union i_u {
    struct mount *i_mton; /* pntr to mount table entry */
    /* that this inode is "mounted on" */
    struct stdata *i_sp; /* Associated stream. */
    struct iisem *iisem; /* ptr to XENIX semaphores */
    struct iisd *iisd; /* ptr to XENIX shared data */
} i_un;
#ifdef FSPTR
    struct fstypsw *i_fstyp; /* pointer to file system */
    /* switch structure */
#endif

    long *i_filocks; /* pointer to filock (structure) list */
    struct rcvd *i_rcvd; /* receive descriptor */
    unsigned long i_vcode; /* inode version code */
    /* (RFS caching) */
    ushort i_went; /* open for write count (RFS caching) */
} inode_t;
extern struct inode inode[]; /* The inode table itself */
struct ifreelist
{ int pad[2]; /* must match struct inode ! */
    struct inode *av_forw;
    struct inode *av_back;
};
extern struct ifreelist ifreelist;
#define i_sptr i_un.i_sp
#define i_mnton i_un.i_mton
#define i_sem i_un.iisem /* for i_ftype==IFNAM && */
/* i_namtype==IFSEM */
#define i_sd i_un.iisd /* for i_ftype==IFNAM && */
/* i_namtype==IFSHD */
/* for IFNAM type files, the subtype is encoded in i_rdev */
#define i_namtype i_rdev

```

```

/* flags */
#define ILOCK      0x01
#define IUPD       0x02
#define IACC       0x04
#define IMOUNT     0x08
#define IWANT      0x10
#define ITEXT      0x20
#define ICHG       0x40
#define ISYN       0x80
/* #define IADV      0x100
#define IDOTDOT    0x200
#define IRMOUNT    0x800
#define IISROOT    0x1000
#define IWROTE     0x200
#define IXLOCKED   0x4000

/* inode is locked */
/* tile has been modified */
/* inode access time to be updated */
/* inode is mounted on */
/* some process waiting on lock */
/* inode is pure text prototype */
/* inode has been changed */
/* do synchronous write for iupdate
/* advertised */
/* object of remote mount */
/* remotely mounted */
/* This is a root inode of an fs */
/* write has happened since open */
/* enforce file locks */
/* for XENIX compatibility */

/* file types */
/* WARNING: The following defines should NOT change! */
/* If more */
/* file types need to be added they should be added */
/* in the low */
/* bits */
#define IFMT      0xf000
#define IFDIR     0x4000
#define IFCHR     0x2000
#define IFBLK     0x6000
#define IFREG     0x8000
#define IFMPC     0x3000
#define IFMPB     0x7000
#define IFIFO     0x1000
#define IFNAM     0x5000

/* type of file */
/* directory */
/* character special */
/* block special */
/* regular */
/* multiplexed char special */
/* multiplexed block special */
/* fifo special */
/* special named file - subtype */
/* in r_dev */
/* XENIX semaphore subtype */
/* of IFNAM type file */
/* XENIX shared-data subtype of IFNAM */

#define IFSEM     1
#define IFSHD     2

/* type file */

/* file modes */
/* the System V Rel 2 chmod system call only knows about */
/* ISUID, ISGID, ISVTX */
/* Therefore, the bit positions of ISUID, ISGID, and ISVTX */
/* should not change */
#define ISUID     0x800
#define ISGID     0x400
#define ISVTX     0x200

/* set user id on execution */
/* set group id on execution */
/* save swapped text even after use

```

```

*/
/* access requests */
/* the System V Rel 2 chmod system call only knows about */
/* IREAD, IWRITE, IEXEC */
/* Therefore, the bit positions of IREAD, IWRITE, and IEXEC */
/* should not change */
#define IREAD    0x100          /* read permission */
#define IWRITE   0x080          /* write permission */
#define IEXEC    0x040          /* execute permission */
#define ICDEXEC  0x020          /* cd permission */
#define IOBJEXEC 0x010          /* execute as an object file */
                                   /* i.e., 410, 411, 413 */
#define IMNDLCK  0x001          /* mandatory locking set */
#define MODEMSK  0xffff        /* Nine permission bits-
read/write/ */
                                   /* execute for user/group/others
and */ /* tSUID, ISGID, and
ISVTX */
/* This is another way of saying:
*/
/* (ISUID|ISGID|ISVTX| */
/* (IREAD|IWRITE|IEXEC)| */
/* ((IREAD
|IWRITE|IEXEC)>>3) */
/* ((IREAD|IWRITE|IEXEC)>>6))
*/
#define PERMMSK  0xffff        /* Nine permission bits: */
                                   /* «IREAD |IWRITE|IEXEC)| */
                                   /* ((IREAD
|IWRITE|IEXEC)>>3) */
                                   /* ((IREAD
|IWRITE|IEXEC)>>6) */
#endif
/* _FS_XX_INODE_H */

```

Найважливішою частиною i-вузла (індексного дескриптора) є список адрес на диску. Це 13 номерів блоків на диску, що зберігаються в індексному дескрипторі. Перші 10 номерів задають перші 10 блоків файлу. Довжина блоку, яку використовує файлова система, може бути різною, але часто складає 1024 байти в блоці. Якщо дані файлу вміщуються в перших 10 блоках чи навіть не займають їх всі, то лише кілька елементів списку містять дискові адреси. Так, якщо файлу досить чотирьох блоків, то перші 4 елементи списку включають відповідні адреси, а інші 9 заповнені нулями. У тому випадку, коли довжина даних перевищує 10 блоків (10240 байтів), перших 10 елементів списку посилаються на блоки файлу, а 11-й - на блок, де розміщується список

наступних 256 номерів блоків. Цей блок називають непрямым. Це зручно для файлів чий розмір не перевищує $10 + 256$ (272 384 байта). Якщо файл менший ніж 266 блоків, 12-й блок містить адресу, що включає адреси 256 номерів блоків. Кожний з них, у свою чергу, містить 256 номерів блоків, що використовуються для знаходження вмісту файла. Тринадцятий блок використовується подібним же способом, тільки вже на наступному рівні. Таким чином, у 12-му блоці міститься адреса подвійного непрямого блоку, а в 13-му блоці - адреса потрійного непрямого блоку.

Якщо перерахувати, то можна знайти, що максимальний розмір одного файла в ОС UNIX складає 16842762 блоки або 17246988288 байтів! Тому, файлова система UNIX накладає обмеження на файли розміром не більше 1 чи 2 Мбайт, щоб випадково не створити файл, який займе всі блоки на диску.

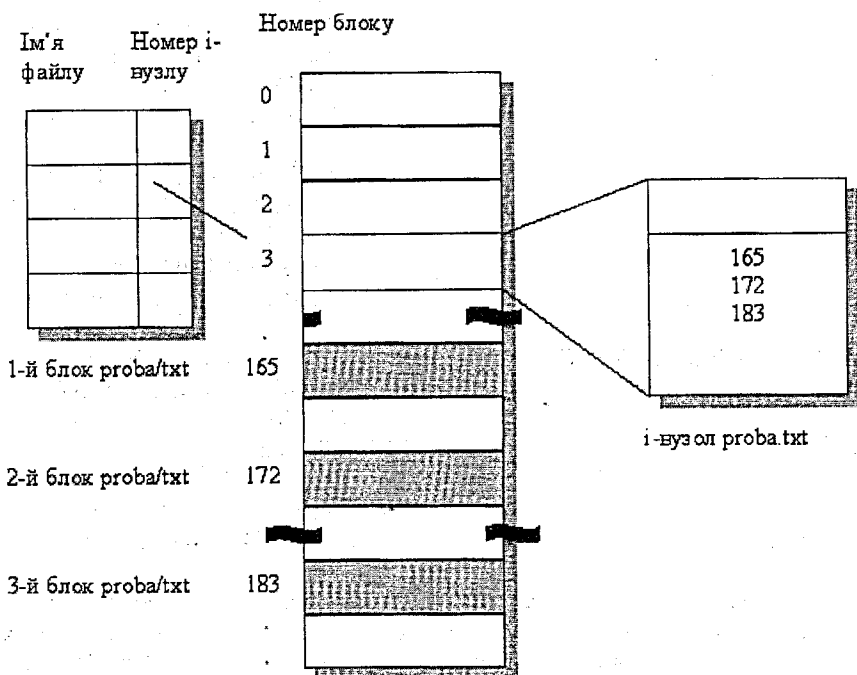


Рис. 2.5. Схема простежування доступу до імені proba.txt у поточному каталозі

Спосіб, за яким файлова система переводить імена файлів в індексні дескриптори, досить простий. Каталог представляє собою файл, що містить таблицю з інформацією. Для кожного файла в каталозі існує рядок у таблиці, що містить ім'я файла і номер індексного дескриптора, який асоціюється з файлом. Коли вводили команду `cat proba.txt`, щоб

одержати вміст файлу на моніторі, файлова система починає насамперед шукати в таблиці поточного каталогу рядок з ім'ям `proba.txt`, знаходить номер `i`-вузла (індексного дескриптора), пов'язаного з цим файлом. Після цього система починає читати блоки, що містять інформацію, записану у файл (рис. 2.5.).

Оцінюючи ефективності файлової системи UNIX необхідно пам'ятати, що після відкриття файлу відповідний індексний дескриптор зчитується в пам'ять `i`, таким чином, системі стають доступні всі номери блоків даного файлу.

Таким чином, кешування диска і збереження в пам'яті копії індексних дескрипторів активних файлів істотно підвищують ефективність файлової системи. Однак є одна істотна обставина, що вкрай небажана для файлових систем UNIX, це раптове відключення живлення. Якщо система працює без UPS (безперебійне джерело живлення), це може призвести до небажаних наслідків, наприклад до втрати блоків, появи некоректних індексних дескрипторів, каталогів і т.і.

Щоб уникнути небажаних наслідків втрати зв'язку системи з диском, ОС UNIX виконує регулярне копіювання резидентних структур даних на диск. Всі системні параметри періодично оновлюються. Крім того, деякі файлові системи ведуть спеціальний журнал і мають здатність до самовідновлення.

2.8. Системні виклики мережевих операційних систем для роботи з файлами

До основних системних функцій, що забезпечують звертання до вже існуючих файлів, відносять `open`, `read`, `write`, `lseek` і `close`. Другими, за важливістю у використанні, стоять функції створення нових файлів - `create` і `mknod`. Далі, - за популярністю у використанні йдуть функції для роботи з індексами файлової системи та зміни привілеїв користувача: `chdir`, `chroot`, `chown`, `stat` і `fstat`.

Як вже відзначалося, простота і потужність системи введення/виведення в ОС UNIX характеризується насамперед тим, що програміст зовсім нічого не зобов'язаний знати про фізичну природу зовнішніх пристроїв. Тому, він оперує тільки з об'єктами-файлами. З погляду користувача знижується різниця між файлом-пристроєм і файлом - набором записів, тому що алгоритм роботи з ними однаковий. Цей підхід дуже нагадує таку ж революційну ідею фірми DEC, що у свій час вразила увагу програмістів: мається на увазі відмова від спеціальних команд введення/виведення в процесорі PDP-7. При цьому, щоб переслати дані з пам'яті на зовнішній пристрій, можна було використовувати звичайну команду `fflow`. Тому всі регістри зовнішніх пристроїв були виведені на загальний пул пам'яті. Але в ОС UNIX для того, щоб вказати ядру файл, до якого необхідно звернутися, треба його або відкрити (`open`), або створити (`create`) (рис. 2.6.).

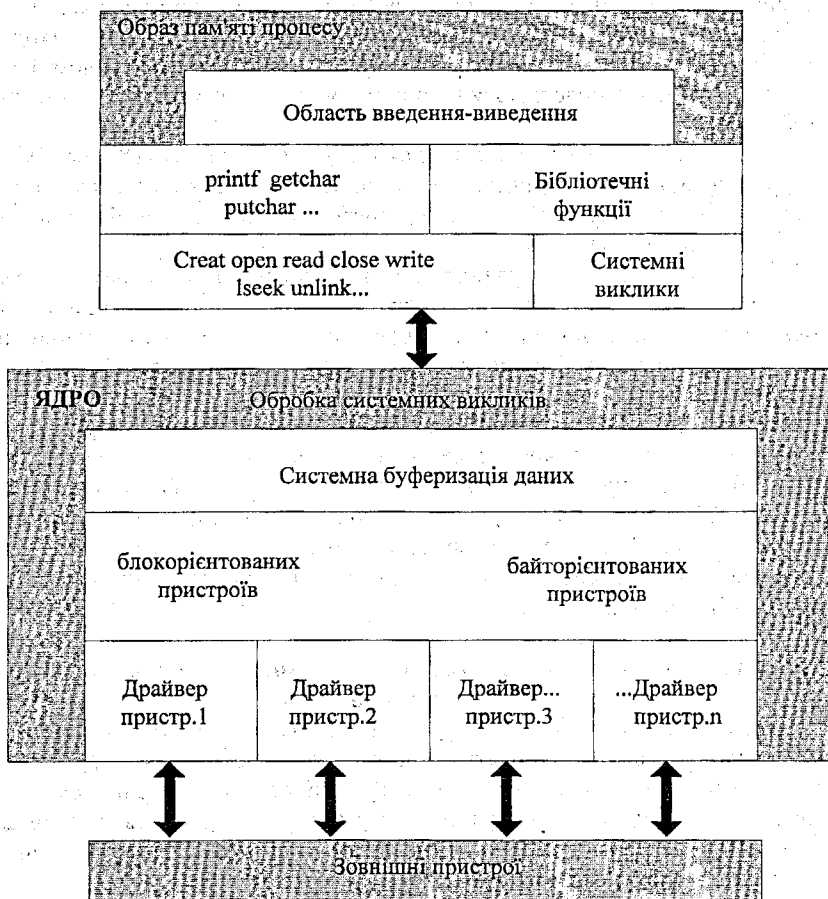


Рис. 2.6. Схема обробки системних викликів

2.8.1. Функція відкриття файла (open)

Відкриття – це процес присвоєння за іменем файла значення деякій змінній, що зберігається в області пам'яті процесу, і названої дескриптором файла. Якщо файл є, існує і його дескриптор та посилання по такому імені дозволене. Синтаксис виклику функції `open` має вигляд:

```
fd = open(pathname, flags, modes);
```

де `pathname` – повне ім'я файла;

`flags` указує режим відкриття (наприклад, для читання чи запису);

`modes` містить права доступу до файла у випадку, якщо файл створюється.

Системна функція `open` повертає ціле число, іменоване користувальницьким дескриптором файлу користувача. Інші операції над файлами, такі, як читання, запис, позиціонування головок диска читання/запису, відтворення дескриптора файлу, установка параметрів введення/виведення, визначення статусу файлу і закриття файлу – використовують для визначення дескриптора файлу, що повертається системною функцією `open`.

Наприклад:

Процес відкриває файл із заданим ім'ям

```
char fname[]="myf.c" /*ім'я файлу */
int fd; /*дескриптор файлу */
char err2[] = "Файл не знайдений /n";
char err13[] = "Порушення привілеїв /n";
char err23[] = "Переповнення таблиці системи /n";
char err24[] = "Занадто багато відкритих файлів /n";
char erru[] = "Невідома помилка * /n";
main()
{
    extern int errno;
    char *p; /*Показчик тексту помилки*/
    if((fd = open(fname,0)) < 0) { /*Спроба відкриття*/
        switch(errno) { /*Обробка помилок*/
            case 2:
                p==err2;
                break;
            case 13:
                p=err13;
                break;
            case 23:
                p=err23;
                break;
            case 24:
                p=err24;
                break;
            default:
                p=erru;
                break;
        }
        write (1,p,30); /*Виведення повідомлення*/
        exit();
    }
    printf("Файл нормально відкритий /n");
    exit();
}
```

2.8.2 Функція створення файлу (create)

Системний виклик create створює новий файл чи перезаписує вже існуючий. Якщо файла не існувало, йому присвоюється код захисту, визначений у параметрі mode. Якщо файл існував, його код захисту та ім'я власника зберігаються, але його довжина обнулюється. Одночасно файл відкривається на запис і користувачу повертається номер дескриптора. Формат виклику create такий:

create (pathname, mode)

де pathname – повне ім'я файла;

mode – містить права доступу до файла у випадку, якщо файл створюється.

Наприклад:

Процес створює файл із заданим ім'ям

```
char myname[] "nef_myf.c";
int fd;
main()
{
    if((fd = create(myname,0666)) < 0) { /*Спроба створення файла*/
        perror("Error create new_myf.c"); /*з кодом захисту 666*/
        exit();
    }
    /* Запис даних у новий
    файл
    */
    close(fd);
}
```

2.8.3 Функція роботи з дескриптором відкритого файла (dup)

Системний виклик dup повертає синонім наявного дескриптора відкритого файла. Звичайно цей виклик використовується, коли необхідно перепризначити введення/виведення. Наприклад, стандартне введення завжди можна підмінити введенням з будь-якого файла.

Нижче наведений приклад показує функції dup.

```
#include <stdio.h>
int fdnew;
int c;
main()
{
    if((fdnew=dup(0))<0) { /* Дублювання дескриптора 0*/
        perror("Error Example");
        exit();
    }
    close(0); /*Закриваємо стандартне введення*/
    open ("myfdat",0); /*Перепризначаємо введення на */
}
```

```

/* введення з файла*/
while((c=getchar())!=EOF putchar(c);
write(fdnew,"end\n",4);
}

```

2.8.4. Функція читання (read)

Функція read служить для обміну інформацією між пам'яттю процесу і зовнішнім об'єктом, коли його дескриптор вже отриманий.

Синтаксис виклику системної функції read:

```
number = read(fd,buffer,count)
```

де fd – дескриптор файла, що повертається функцією open;

buffer – адреса структури даних у користувацькому процесі, де будуть розміщуватися зчитані дані у випадку, якщо успішно завершиться виконання функції read;

count – кількість байтів, що користувачу потрібно прочитати;

number/кількість фактично прочитаних байтів.

Після того, як ядро встановить значення параметрів введення/виведення в адресному просторі процесу, воно звертається до індексу, використовуючи покажчик з таблиці файлів, і блокує його перш, ніж почати читання з файла. При цьому в алгоритмі обробки даних починається цикл, що виконується доти, поки операція читання не буде оброблена до кінця. Ядро здійснює зсув у байтах всередині файла в номер блоку, використовуючи алгоритм bmap. Потім обчислює зсув усередині блоку до місця, звідки треба почати введення/виведення і кількість байтів, що будуть прочитані з блоку.

Після зчитування блоку в буфер, за допомогою алгоритмів bread і breada, ядро копіює дані з блоку за адресою, встановленою в користувацькому процесі. Якщо запит користувача не виконаний, ядро повторює весь цикл: перетворює зсув у байтах з середини файла в номер блоку; зчитує блок з диска в системний буфер; виконує копію даних з буфера в користувацький процес; звільняє буфер і коректує значення параметрів введення/виведення в адресному просторі процесу.

Цикл завершується якщо:

- 1) ядро виконає запит користувача повністю;
- 2) у файлі скінчилися дані;
- 3) ядро знаходить помилку при читанні даних з диска або при копіюванні даних у простір користувача.

Ядро коректує значення зсуву в таблиці файлів відповідно до кількості фактично прочитаних байтів. Тому успішне виконання операцій читання виглядає як послідовне зчитування даних з файла:

```

#include <fcntl.h>
main()
{
    int fd;
    char lbuf[20],bigbuf[1024];

```

```

fd = open("/etc/hosts", O_RDONLY);
read(fd, lbuf, 20);
read(fd, bigbuf, 1024);
read(fd, lbuf, 20);
}

```

2.8.5. Функції захоплення процесів (fcntl) та записів (write)

У перших версіях ОС UNIX був відсутній механізм, за допомогою якого процес міг забезпечити собі винятковий доступ до файла. Незважаючи на протести розробника мови високого рівня C – Річі, у версію System V були включені механізми захоплення файла і запису. Захоплення файла – це засіб, що дозволяє заборонити іншим процесам робити читання чи запис будь-якої частини файла. Захоплення запису – це спосіб, що дозволяє заборонити іншим процесам робити введення/виведення зазначених записів. У версії System V функція fcntl використовується для реалізації механізму захоплення файла і запису і має такий формат:

```
fcntl(fd, cmd, arg);
```

де fd – дескриптор файла;

cmd – тип операції, що блокує, а в arg вказуються різні параметри, такі як тип блокування (запису чи читання) і зсуву в байтах;

arg – аргумент, який задається у вигляді константи.

Параметри cmd і arg інтерпретуються в такий спосіб (визначення констант зберігається у файлі "/usr/include/fcntl.h"):

F_DUPFD повернути найменше значення дескриптора, більше чи рівне значенню arg;

F_SETFD встановити прапорець "close-on-exec" у молодшому розряді arg (файл буде закритий функцією exec);

F_GETFD повернути стан прапорця "close-on-exec";

F_SETFL установити прапорці, що керують станом файла (O_NDELAY – продовжувати працювати не чекаючи завершення введення/виведення, O_APPEND – дані додавати в кінець файла);

F_GETFL одержати значення прапорців, що керують станом файла:

STRUCT

FLOCK
short l_type;

short

l_whence;

long l_start;

F_RDLCK – блокування читання, **F_WRLCK** – блокування запису, **F_UNLCK** – зняття блокування; адреса початку блокованої ділянки дається у вигляді зсуву відносно початку файла (0), щодо поточної позиції покажчика (1), щодо кінця файла (2); зсув у байтах, інтерпретований у відповідності зі значенням l_whence;

<code>long l_len;</code>	довжина блокованої ділянки в байтах. Якщо зазначений 0, блокується ділянка від <code>l_start</code> до кінця файла;
<code>long l_pid;</code>	ідентифікатор процесу, що блокує файл;
<code>long l_sysid;</code>	системний ідентифікатор процесу, що блокує файл;
<code>F_GETLK</code>	прочитати перший код блокування, що заважає використовувати значення <code>arg</code> і знищувати його. Якщо блокування відсутнє, поміняти значення <code>l_type</code> в <code>arg</code> на <code>F_UNLCK</code> ;
<code>F_SETLK</code>	встановити чи зняти блокування файлу в залежності від значення <code>arg</code> . У випадку неможливості встановити блокування повернути -1;
<code>F_SETLKW</code>	встановити чи зняти блокування даних, що містяться у файлі, в залежності від значення <code>arg</code> . У випадку неможливості встановити блокування призупинити виконання.

Процес блокування подібний до методу транзакцій. Однак, при використанні деяких процесів водночас існують деякі відмінності. Наприклад, блокування, які пов'язані з читанням з файлу, можуть перекривати один одного. Блокування, які пов'язані з записом, перекриватися не можуть.

Функція запису є дуже поширеною у програмах користувача. Вона не є складною і зустрічається досить часто в цьому підручнику.

Синтаксис виклику системної функції `write` такий:

```
number = write(fd,buffer,count);
```

де змінні `fd`, `buffer`, `count` і `number` мають той же зміст, що і для виклику системної функції `read`.

Алгоритм запису в звичайний файл схожий на алгоритм читання з файлу, за винятком того, що дескриптор повинен бути попередньо створений або файл відкритий для запису. Практично система не обмежує довжину переданої інформації, однак при записі на диски і стрічки найкраще працювати блоками. Ці операції проводяться з більшою ефективністю. При обробці символьних даних, коли виникає необхідність приймати і передавати дані з файлу у файл по одному символу, можна використовувати системні функції `getchar` і `putchar`.

2.8.6 Функція знаходження необхідної позиції у файлі (`lseek`)

Системний виклик `lseek` забезпечує просування покажчика до потрібного місця у файлі для здійснення операцій введення/виведення.

Синтаксис виклику системної функції:

```
position = lseek(fd,offset,ptr);
```

де `fd` – дескриптор файлу, що ідентифікує файл;

`offset` – зсув у байтах;

ptr – вказівник значення offset, який має значення:
зсув від початку файлу - 0,
зсув від поточної позиції введення/виведення - 1;
зсувом від кінця файлу - 2.

Значення, що повертається у змінну position, є зсувом у байтах до місця, де буде починатися наступна операція читання чи запису. Наприклад, у програмі, наведеній нижче, процес відкриває файл, зчитує байт, а потім викликає функцію lseek, щоб замінити значення зсуву в таблиці файлів розміром у 1023 байта. Змінна reference, має значення 1.

Наступний приклад демонструє процес пошуку позиції, що задана в програмі:

```
#include <fcntl.h>
main(argc,argv)
int argc;
char *argv[];
{
    int fd,skval;
    char c;
    if(argc != 2)
        exit();
    fd = open(argv[1],O_RDONLY);
    if (fd == -1)
        exit();
    while ((skval = read(fd,&c,1)) == 1)
    {
        printf("символ: %c\n",c);
        skval = lseek(fd,1023L,1);
        printf("пошук символу знайдений в позиції: %d\n",skval);
    }
}
```

Таким чином, програма зчитує кожен 1024-й байт файлу. Якщо reference має значення 0, ядро здійснює пошук від початку файлу, якщо 2 – проводиться пошук від кінця файлу. Функція lseek нічого не повинна робити, крім операції пошуку, що позиціонує головку диска читання/запису на зазначений дисковий сектор. Для того, щоб виконати функцію lseek, ядро вибирає значення зсуву з таблиці файлів. У подальших викликах функцій read і write вибір позиції з таблиці файлів використовується як початковий зсув.

2.8.7. Функція закриття файлу (close)

Функція закриття файлу є також найпростішою та найпоширенішою у використанні в навчальних та професійних програмах. Формат системного виклику close такий:

```
close (fd)
```

де fd – дескриптор файла, що закривається.

Закриття файлів, відкритих процесом, відбувається автоматично при завершенні процесу. Тому існує обмеження на кількість одночасно відкритих одним процесом файлів. Наприклад, при закритті файлів стандартного введення і виведення використовується функція close. При цьому, в ОС UNIX код повернення при успішному завершенні операції – 0, в протилежному випадку – 1.

2.8.8. Стандартні бібліотечні функції введення/виведення

ОС UNIX з самого початку розроблялась мобільною платформою зі переносними програмами. Тому всі системні виклики в UNIX оформлені як С-інтерфейси. На базі цих системних викликів у ОС UNIX створена бібліотека функцій для роботи з потоками байтів. Ці функції є потужними. Їх рекомендується використовувати як найчастіше, щоб зробити програми мобільнішими.

Бібліотечні функції введення/виведення розташовуються в бібліотеці "/lib/libc.a". Тому програми, що написані мовою С, автоматично звертаються до цієї бібліотеки при побудові своїх модулів. Якщо ними потрібно скористатися при створенні програм на іншій мові програмування, то необхідно вказати: для компілятора – ключ lc, для компоновщика – ld.

2.8.9. Потоківі функції введення/виведення (getchar, putchar)

Функція getchar повертає процесу черговий байт зі стандартного введення. Функція putchar записує байт на стандартне виведення. Ці функції використовуються в програмах дуже просто. Наприклад, щоб скопіювати без перетворення вхідний файл у вихідний, треба використати конструкцію виду:

```
while((c=getchar()) != EOF)
    putchar( c);
```

2.8.10 Бібліотечні функції доступу до файлів

Для використання бібліотечних функцій, треба обов'язково вказати в програмі включення файла заголовків стандартних функцій <stdio.h>. Для файлів існують спеціальні бібліотечні функції – fopen, fclose і т.і. Також існує спеціальна файлова структура, що описується в програмі за допомогою оператора:

```
FILE *fp, *fopen();
```

де fp – покажчик на FILE, що повертається в результаті виконання функції fopen виду:

```
fp=fopen(name,mode)
```

Перший аргумент – це, покажчик на ім'я файла, а другий аргумент задає режим відкриття: r (читання), w (запис), a (доповнення). Якщо файл, що відкривається, не існує, то при відкритті для запису і доповнення він створюється автоматично. Якщо файл існував, то старий зміст

його при записі губиться. Приведемо приклад невеликої програми, що з одного вхідного файла створює два вихідних.

```
#include <stdio.h>
#define mes1 "помилка відкриття файла\n"
#define mes2 "робота виконана\n"
FILE *fp1, *fopen();
FILE *fp2;
FILE *fp3;
int c;
main()
{
    if((fp1=fopen("my.txt","r")) == NULL) {
        printf(mes1);
        exit();
    }
    if((fp2=fopen("dup1.txt","w")) == NULL) {
        perror("помилка у записі файла");
        exit();
    }
    if((fp3=fopen("dup2.txt","w")) == NULL) {
        perror("помилка у записі файла");
        exit();
    }
    while((c=sgetc(fp1)) != EOF) {
        putc(c,fp2);
        putc(c,fp3);
    }
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    printf(mes2);
}
```

Приклад 2.8.10.1

Важливо, що всі ці найпростіші програми на C з однаковим успіхом працюють і в UNIX і в MS DOS, транлюються без помилок майже будь-яким транслятором C. Їх можна побудувати і виконати в ОС UNIX таким чином:

Команда	Примітка
cc dup.c	компіляція файла
./a.out	виконання
або в MS DOS:	
cl dup.c	компіляція файла
dup.exe	виконання

Ще однією потужною та такою, що легко переноситься на різні платформи мовою є Java, яка буде детально описана в наступній главі.

Контрольні питання

1. Розподіл дискового простору. Загальна характеристика області завантаження, приклади. Найменування та призначення перших блоків.
2. Поняття про індексні дескриптори. Відмінності між логічним та фізичним представленням файла.
3. Побудуйте загальну схему доступу до будь-якого імені файла, надайте стислу характеристику цьому процесу.
4. Системні виклики, призначення, класифікація, методи роботи.
5. Призначення та основні характеристики функції відкриття файлів (open) на мові високого рівня С.
6. Призначення та основні характеристики функції створення файлів (create) на мові високого рівня С.
7. Призначення та основні характеристики функції роботи з дескрипторами відкритого файла (dup) на мові високого рівня С.
8. Призначення та основні характеристики функції читання файлів (read) на мові високого рівня С.
9. Призначення та основні характеристики функції запису файлів (write) на мові високого рівня С.
10. Призначення та основні характеристики функції знаходження необхідної позиції у файлі (lseek) на мові високого рівня С.
11. Призначення та основні характеристики функції закриття файлів (close) на мові високого рівня С.
12. Загальна характеристика бібліотечних функцій введення/виведення на мові високого рівня С. Поточкові функції введення/виведення, приклади.
13. Загальна характеристика бібліотечних функцій введення/виведення на мові високого рівня С. Бібліотечні функції доступу до файлів, приклади.

Контрольні вправи

1. В ОС Unix знайдіть файл, який відповідає за структуру індексних дескрипторів файлової системи. Охарактеризуйте деякі важливі структури міжпроцесних комунікацій.
2. Напишіть програму "myprog.c" на мові високого рівня С, яка тестує наявність файла на жорсткому диску. Визначте діагностику основних можливих помилок читання файла.
3. Напишіть програму "myprog.c" на мові високого рівня С, яка створює файл з кодом захисту 0555. Поясніть значення цього коду.
4. Напишіть програму "myprog.c" на мові високого рівня С, яка взаємодіє з дескрипторами відкритого файла.
5. Напишіть програму "myprog.c" на мові високого рівня С, яка читає та виводить зміст файла "myprog.txt".
5. Напишіть програму "myprog.c" на мові високого рівня С, яка записує інформацію в файл "myprog.txt".

6. Напишіть програму "myprog.c" на мові високого рівня C, яка знаходить необхідну позицію (задану користувачем) у файлі "myprog.txt".

2.9. Приклади програмування на Java для роботи з файловими мережевими системами. Файлові об'єкти

Для одержання інформації щодо файлів чи каталогів, використовуються об'єкти File. Об'єкти File повинні відрізнятися від об'єктів потоку типу OutputStream і InputStream. Об'єкт File використовується, щоб одержати інформацію щодо специфічного файла чи каталогу, і не використовується для читання і запису даних.

Наприклад, наступна прикладна програма створює об'єкт File для специфічного файла і задає при цьому кілька питань. Прикладна програма одержує ім'я файла як параметр командного рядка.

```
import java.applet.*;
import java.io.*;
public class FileObjectTest
{
    public static void main(String[] args)
    {
        if (args.length <= 0 )
        {
            System.err.println("неправильно вказаний аргумент програми");
            return;
        }
        String fileName = args[0];
        File f = new File(fileName);
        if ( !f.exists() )
        {
            System.err.println("не можу знайти " + fileName);
            return;
        }
        System.out.println("Файл " + fileName + " є " +
            f.length() + " байт довжини");
        if (f.isDirectory())
            System.out.println("вказане ім'я " + fileName + " є каталогом");
        else
            System.out.println("Файл " + fileName +
                " не є каталогом");
    }
}
```

Програма 2.9.1

2.9.1. Функції виведення записів

Java забезпечує спосіб запису байтів на FileOutputStream. Це виконується за допомогою запису одиниць примітивних типів.

Наприклад для такого типу даних як рухома крапка на `DataOutputStream` і різних об'єктів типу `PrintStream`.

Інші об'єкти – `FileOutputStream` та `DataOutputStream` являють собою потік байтів, які записують одиниці примітивного типу в двійковій формі. Другий клас – `PrintStream` друкує коди в ASCII або UNICODE. Ці 3 класи найчастіше використовуються в програмах користувачів. Всі вони створюються на базі імені `String` або файла з об'єкта `File`.

Наприклад, клас `FileOutputStream` може бути використаний таким чином:

```
import java.awt.*;
import java.io.*;
public class FileOutputStreamTest
{
    public static void main(String[] args) throws IOException
    {
        char[] someBytes = {'a', 'b', 'c'};
        String hello = "hello";
        FileOutputStream f=null;
        f = new FileOutputStream("junk");
        if (f!=null)
        {
            for (int i=0; i<3; i++)
                f.write((int)someBytes[i]);
            for (int i=0; i<hello.length(); i++)
                f.write((int)hello.charAt(i));
            f.close();    // закрити та звільнити системні ресурси
        }
    }
}
```

Програма 2.9.1.1

Програма 2.9.1.1 демонструє виведення потоку даних у файл.

Наступна програма показує потоковий спосіб виведення інформації:

```
import java.awt.*;
import java.io.*;
public class DataOutputStreamTest
{
    public static void main(String[] args) throws IOException
    {
        int i = 34;
        String hello = "hello";
        FileOutputStream f = null;
        DataOutputStream df = null;
        // make a generic output byte stream
        f = new FileOutputStream("junk");
        // приєднання класу DataOutputStream до потоку даних
        df = new DataOutputStream(f);
    }
}
```

```

if (df!=null)
{
    df.write(i);
    df.writeBytes(hello);
    df.close(); // закриття та звільнення системних ресурсів
    f.close(); // закриття файлового потоку даних
}
}
}

```

Програма 2.9.1.2

Наступний програма також реалізує виведення потоку даних, але за допомогою класу `PrintStream`.

```

import java.awt.*;
import java.io.*;
public class PrintStreamTest
{
    public static void main(String[] args) throws IOException
    {
        FileOutputStream f = null;
        PrintStream pf = null;
        Button anArbitraryObject = new Button("a button");
        // створення потоку виведення даних
        f = new FileOutputStream("junk");
        // приєднання класу PrintStream до виведення даних
        pf = new PrintStream(f);
        if ( pf!=null)
        {
            pf.println(34);
            pf.println("тестовий рядок " + 34);
            pf.println(3.14159);
            pf.println("Це тестове повідомлення");
            pf.println(anArbitraryObject);
            pf.close(); // закриття та звільнення системних ресурсів
            f.close(); // закриття файлового потоку даних
        }
    }
}

```

Програма 2.9.1.3

Таким чином можна підвести підсумок, що клас `FileOutputStream` записує тільки байти, `DataOutputStream` записує примітивні одиниці і рядки в простому двійковому вигляді. Клас `PrintStream` виводить примітивні одиниці та об'єкти у вигляді текстових файлів – тобто ціле число 34 буде записано як два символи: 3 і, наступна, 4 (на відміну від чотирьох байтового двійкового слова зі значенням 34).

2.9.2. Буферизоване виведення інформації

Клас `BufferedOutputStream` дозволяє направляти потік байтів на виведення без фактичної актуалізації файлової системи при кожній операції звернення. Об'єкти `BufferedOutputStream` повинні бути приєднані до `OutputStream` і потім використовуватися звичайним чином:

```
f = new FileOutputStream("junk");
bf = new BufferedOutputStream(f);
df.write(34);
```

Приклад 2.9.2.1

2.9.3. Клас `FileOutputStream`

В багатьох програмах на Java виконується клас `FileOutputStream`, щоб записувати байти і забезпечувати функціонування "фільтра", який називається `DataOutputStream`. Цей клас підтримує виведення більш складних одиниць.

`DataOutputStream` може бути приєднаний до будь-якого `OutputStream`, щоб забезпечити ці додаткові функціональні можливості. Наприклад, Java не підтримує клас `BufferedDataOutputStream` для виведення складних структур, але забезпечує "фільтри" класів `DataOutputStream` і `BufferedOutputStream`. Ці класи можуть бути приєднані до базисного `FileOutputStream`. Наприклад, програма, яка наведена далі, виконує ці можливості:

```
import java.awt.*;
import java.io.*;
public class BufferedDataOutputStream
{
    public static void main(String[] args) throws IOException
    {
        int i = 34;
        String hello = "hello";
        FileOutputStream f = null;
        BufferedOutputStream bf = null;
        DataOutputStream df = null;
        f = new FileOutputStream("junk");
        bf = new BufferedOutputStream(f);
        df = new DataOutputStream(bf);
        if (df!=null)
        {
            df.write(i);
            df.writeBytes(hello);
            df.close();
            bf.close();
            f.close();
        }
    }
}
```

Програма 2.9.3.1

При записі `DataOutputStream` обробляє байти, що повинні бути записані і посилає їх у `BufferedOutputStream`. `BufferedOutputStream` зберігає байти в буфері, поки він не стає повним. Потім викликається клас `FileOutputStream`, який направляє байти у файлову систему.

Більш "тонке" рішення для ефективного виведення інформації полягає у використанні "делегування". В даному випадку це є більш потужна можливість виведення даних, ніж метод фіксованого спадкування. Делегування використовується, коли функції об'єкта під час його виконання повинні бути змінені. Наприклад, можна використовувати делегування, щоб визначити поведінку об'єкта чи появу частин, що змінюються. Тому клас `BufferedOutputStream` приєднується до `OutputStream` через конструктор і потім делегує велику кількість його функцій. Наприклад, в конструкції, показаний нижче клас `BufferedOutputStream` делегує методи класу `FileOutputStream`:

```
class BufferedOutputStream extends FileOutputStream
```

```
{
    OutputStream out;
    ...
    void write(int byte)
    {
        ...
    }
    ...
}
```

Клас `FileOutputStream` містить посилання до `OutputStream`, що може бути підкласом `FileOutputStream`. Отже, клас `BufferedDataOutputStream` може бути створений під час виконання ланцюжка фільтрів `DataOutputStream`, `BufferedOutputStream` і `FileOutputStream`, зібраних разом.

Наприклад, `OutputStream`-ієрархія виглядає таким чином:

<code>OutputStream</code>	[генерує вихідний потік даних]
<code>FileOutputStream</code>	[має основні функції роботи з файловим потоком даних]
<code>BufferedOutputStream</code>	[має основні функції роботи з буферним потоком]
<code>DataOutputStream</code>	[має функції запису примітивних бінарних образів]
<code>PrintStream</code>	[має функції запису примітивних об'єктів у текстовому вигляді]

2.9.4. Функції операцій читання

Java забезпечує засіб для читання байтів з класу `FileInputStream` а також читання рядків та одиниць примітивного типу (наприклад, тип з рухомою крапкою) з класу `DataInputStream`. Процес читання текстових одиниць і приведення їх до примітивних типів виконується за допомогою символьних змінних (`string`). Потім аналізується рядок за допомогою методів типу `Integer.parseInt()` і `Float.valueOf()`. Тому при читанні даних з файлу використовується клас `FileInputStream` (програма 2.9.4.1), а з клавіатури – клас `DataInputStream` (програма 2.9.4.2).

```
import java.awt.*;
import java.io.*;
public class FileInputStreamTest
{
    public static void main(String[] args) throws IOException
    {
        byte[] buffer = new byte[1024];
        int b;
        FileInputStream f = null;
        f = new FileInputStream("junk");
        if (f!=null)
        {
            b = f.read();
            f.read(buffer);
            f.close();
        } } }
Програма 2.9.4.1
```

```
import java.awt.*;
import java.io.*;
public class DataInputStreamTest
{
    public static void main(String[] args) throws IOException
    {
        int i;
        String str = null;
        FileInputStream f = null;
        DataInputStream df = null;
        f = new FileInputStream("junk");

        if ( df!=null)
        {
            i = df.read();
            str = df.readLine();
            System.out.println("read int " + i + " and " + str);
            df.close(); // close and free up system resource.
        }
    }
}
```

```
f.close(); // close in reverse order; FileInputStream last
```

Програма 2.9.4.2

Наступна програма показує реалізацію класа `StringBufferInputStream` для більш потужного введення даних за допомогою буферного потоку даних:

```
import java.awt.*;
import java.io.*;
public class StringBufferInputStreamTest
{
    public static void main(String[] args) throws IOException
    {
        String junk = "abc34def";
        byte[] buffer = new byte[1024];
        int b;
        StringBufferInputStream f = null;
        f = new StringBufferInputStream(junk);
        if (f!=null)
        {
            b = f.read();
            // buffer will be {'b','c','3','4','d','e','f'}
            f.read(buffer);
            f.close();
        }
    }
}
```

Програма 2.9.4.3

2.9.5. Клас `FileInputStream`.

Як і `FileOutputStream`, `FileInputStream` використовує делегування, щоб дозволити `InputStream`-об'єктам бути зв'язаними в ланцюжок. Клас `FileInputStream` містить посилання до `InputStream`, з яким його функції можуть бути делеговані. Використання делегування дозволяє новим комбінаціям функцій бути створеними під час виконання. В той же час спадкування – це механізм, який працює тільки на етапі компіляції. Тому він вважається не таким гнучким.

Наприклад, клас `BufferedDataInputStream` може бути створеним під час формування ланцюжка фільтрів `DataInputStream`, `BufferedInputStream` і `FileInputStream`, зібраних разом.

Клас `InputStream` має таку ієрархію:

<code>InputStream</code>	[генерує базисний клас потоку вхідних даних]
<code>FileInputStream</code>	[генерує базисний клас потоку вхідних файлових]

даних]

DataStream [функції читання примітивних типів даних]

BufferedInputStream [функції буферного введення даних]

StringBufferInputStream [функції буферного введення даних типу рядок]

2.9.6. Операції довільного доступу до файлів

Часто необхідно читати чи писати дані, що зберігаються у файлах, довільним методом доступу, а не послідовно, як у випадку з магнітною стрічкою. Java забезпечує клас `RandomAccessFile`, що поводить себе подібно об'єднаному `DataOutputStream` і `DataStream` (у `RandomAccessFile` застосовуються `DataOutput` і `DataInput`).

Об'єкти `RandomAccessFile` використовують ім'я файла чи об'єкта `File`, подібно іншим об'єктам потоку, але параметр конструктора режиму ім також потрібно. Для режимів роботи з файлами використовується `r` (читання) чи `rw` (читання/запис) точно так само, як у функції `open()` у C чи C++. Наприклад:

```
RandomAccessFile f = new RandomAccessFile("junk", "rw");
```

```
File fo = new File("crud");
```

```
RandomAccessFile f = new RandomAccessFile(fo, "r");
```

Щоб додати інформацію до `RandomAccessFile`, необхідно знайти кінець файла:

```
f.seek(f.length());
```

Щоб почати читання файла з позиції `position` необхідно знайти початок файла, використовуючи функцію:

```
f.seek(0);
```

Наступна прикладна програма пише дані у файл і потім знову читає його спочатку:

```
import java.awt.*;
```

```
import java.io.*;
```

```
public class RandomAccessTest
```

```
{
```

```
    public static void main(String[] args) throws IOException
```

```
{
```

```
    RandomAccessFile f = new RandomAccessFile("junk", "rw");
```

```
    if (fb==null )
```

```
{
```

```
        f.write(34);
```

```
        f.writeDouble(3.14159);
```

```
        f.writeBytes("Hello");
```

```
        f.seek(0);
```

```
        int i = f.read();
```

```
        double g = f.readDouble();
```

```
        string str = f.readLine();
```

```
        if (i!=34 || g!=3.14159 || !str.equals("Hello"))
```

```
            System.out.println(" Проблеми з записом до файлу");
```

```

        f.close();
    }
}

```

Програма 2.9.6.1

2.9.7. Методи читання і запису даних до мережевих файлових систем

Файли на мережевих файлових системах можуть читатися/писатися через гнізда – sockets або при використанні об'єктів URL. Користувач може запросити об'єкти за допомогою функції InputStream для одержання можливості читати з них.

Наприклад:

```

import java.awt.*;
import java.io.*;
import java.net.*;
public class URLAccessTest
{
    public static void main(String[] args) throws IOException
    {
        byte[] buffer = new byte[100];
        in = new URL("ftp://ftp.vstu.vinnica.ua/welcome.msg").openStream();
        if (in!=null)
        {
            in.read(buffer);
            String s = new String(buffer, 0);
            System.out.println("first <=100 chars are: " + s);
            in.close();
        }
    }
}

```

Програма 2.9.7.1

У програмі виконується друкування файла, що знаходиться на міжнародному сервері з ім'ям ftp.vstu.vinnica.ua . Це реальний сервер, що знаходиться у Вінницькому технічному університеті. Доступ до нього здійснюється для будь-якого анонімного користувача. Щоб здійснити доступ до захищеного сервера, потрібно застосовувати спеціальні конструктори.

2.10 Мережеві комунікації операційних систем

Java може використовуватися для зв'язку з мережевими файловими системами, що застосовують модель "клієнт-сервер". Сервер обробляє запити на вхідні з'єднання від клієнтів у межах усієї мережі або окремої машини. Клієнти знають як з'єднатися із сервером через установлений номер порту. Після з'єднання сервер породжує потік, щоб обробити

запит, який приходить з "каналу" від клієнта. Таким чином, велика кількість клієнтів можуть з'єднуватися з тим же самим сервером.

На деяких прикладах розробимо декілька програм. Першу програму назовемо SocketServer. До важливих її функцій віднесемо приєднання до специфічного номера порту (32-розрядне ціле число відоме всім клієнтам) і очікування запитів на з'єднання від клієнтів. При цьому сервер і користувачі будуть одержувати параметри socket у результаті з'єднання. Об'єкти Socket – це канали для читання-запису між мережевими ОС, до яких можна звертатися через об'єкти InputStream і OutputStream. Гнізда (sockets) – це унікальні ідентифікатори між різними частинами мережових ОС, тому що складаються з таких чотирьох елементів:

- адреси IP-сервера;
- адреси IP-клієнта;
- номеру порту сервера;
- номеру порту клієнта.

Для кожного запиту клієнта socket призначає унікальний номер порту, у той час як номер порту сервера завжди залишається єдиним.

Дані, що пересилаються по утвореному каналу, можуть належати до різних типів. Можна уявити, що на стороні сервера існує файлова база даних, тобто виконавець файлових запитів віддаленого користувача.

В програмі 2.10.1 та 2.10.2 наведемо базисний процес з'єднання за допомогою сокетів клієнта і сервера.

Спочатку розглянемо реалізацію прикладної програми сервера.

```
import java-awt.*;
import java.io.*;
import java.net.*;
public class SimpleServer
{
    public static void main(String[] args)
    {
        Server serv = new Server(5551212);
        serv.out.println("Відправка тестового повідомлення\n");
        try
        {
            String data = serv.in.readLine();
            System.out.println("Клієнтська частина прийняла " + data);
        }
        catch (IOException e) {}
    }
}
class Server //Сервер очікує з'єднання через встановлений порт
{
    Socket channel;
    public DataInputStream in;
```

```

public PrintStream out;
ServerSocket server;
public Server(int port)
{
    try
    {
        // сервер створює канал з timeout у 15 секунд
        server = new ServerSocket(port, 15);
        channel = server.accept();
        out = new PrintStream(channel.getOutputStream());
        in = new DataInputStream(channel.getInputStream());
    }
    catch (IOException e) {}
}
}

```

Програма 2.10.1

Програма 2.10.1 встановлює зв'язок з програмою-отримувачем з прикладу 2.10.2:

```

import java.awt.*;
import java.io.*;
import java.net.*;
public class SimpleClient
{
    public static void main(String[] args)
    {
        Client client = new Client(" Java-one", 5551212);
        try
        {
            String data = client.in.readLine();
            System.out.println("Server says: " + data);
        }
        catch (IOException e) {}
        client.out.println("Heh, I heard you over this socket!\n");
    }
}

```

// Відкриття каналу для заданого хосту та порту

```

class Client
{
    public DataInputStream in;
    public PrintStream out;
    Socket client;
    public Client(String host, int port)
    {
        try
        {

```

```

        client = new Socket(host, port);
        out = new PrintStream(client.getOutputStream());
        in = new DataInputStream(client.getInputStream());
    }
    catch (IOException e) {}
}

```

Програма 2.10.2

Необхідно пам'ятати, що програма-відправник (Програма 2.10.1) повинна бути виконана перед програмою-отримувачем (Програма 2.10.2).

Транслювати програми на Java необхідно використовуючи програму `javac`, наприклад:

```
javac client.java
```

Виконувати побудовані програми потрібно за допомогою інтерпретатора:

```
java client
```

Контрольні питання

1. Загальна характеристика файлових об'єктів мови високого рівня Java, приклади.

2. Реалізація функцій виведення записів на мові високого рівня Java. Охарактеризуйте на прикладах функції `FileOutputStream`, `DataOutputStream` та `PrintStream`, зробіть їх порівняльну характеристику.

3. Реалізація буферизованого виведення інформації на мові високого рівня Java. Охарактеризуйте на прикладах функції `BufferOutputStream` та `FileOutputStream`, зробіть їх порівняльну характеристику.

4. Реалізація функцій читання інформації на мові високого рівня Java. Охарактеризуйте на прикладах функції `FileInputStream`, `DataInputStream` та `StringBufferInputStream`, зробіть їх порівняльну характеристику.

5. Реалізація операції довільного доступу до файлів на мові високого рівня Java. Наведіть приклади.

6. Реалізація функцій читання та запису даних мережевих файлових систем на мові високого рівня Java. Наведіть приклади.

7. Реалізація мережевих комунікацій на мові високого рівня Java. Наведіть приклади.

Контрольні вправи

1. Напишіть програму `"myprog.java"` на мові високого рівня Java, яка створює файл.

2. Поясніть роботу програми 2.9.1.1.

3. Поясніть роботу програми 2.9.1.2.

4. Поясніть роботу програми 2.9.1.3.

5. Поясніть роботу програми 2.9.3.1.

6. Поясніть роботу програми 2.9.4.1.

7. Поясніть роботу програми 2.9.4.2.
8. Поясніть роботу програми 2.9.4.3.
9. Поясніть роботу програми 2.9.6.1.
10. Поясніть роботу програми 2.9.7.1.
11. Поясніть роботу програми 2.10.1.
12. Поясніть роботу програми 2.10.2.

2.11. Загальна характеристика відомих представників мережових операційних систем

Попередній матеріал мав мету дати короткий загальносистемний опис сучасної мережевої ОС UNIX. Зрозуміло, що в даному матеріалі викладена неповна і спрощена концепція UNIX-системи, переважно для навчального засвоєння. Однак основні компоненти ядра і файлової системи дають деяке уявлення про функціонування ОС UNIX стандартів BSD та SYSTEM V (Solaris).

В даний час архітектура ОС UNIX являє собою втілення ідей, які впроваджувалися протягом великого історичного проміжку часу, з погляду розвитку обчислювальної техніки. Від початку 60-х до середини 90-х років, були розроблені майже всі основні стандарти, якими керуються розробники мережових ОС. Законовизначними для ОС UNIX є прийняті стандарти і рекомендації, розроблені:

- в Міжнародній організації по стандартизації - ISO/OSI (International Standard Organization/Open Systems Interconnection);
 - в Індустріальному консорціумі, спонсорами якого є IBM, Digital, Hewlett Packard, SCO - Open Software Foundation (OSF);
 - в X/Open компанія, що розробила набір стандартів, відомих як Common Applications Environment, що використовуються для проектування мов програмування, баз даних, інтерфейсів користувача інтерфейсів, обчислювального устаткування і локальних мереж - X/OPEN Standards;
 - в комітеті інженерів з електроніки, електротехніки і радіоелектроніки IEEE - стандарт POSIX для розробки програмного забезпечення, а також стандарти, що регламентують сумісність прикладного і програмного забезпечення на рівні розробки графічних інтерфейсів і можливості використання в UNIX додатків, призначених для виконання в середовищі DOS і Windows:
- 1) System V Binary Compatibility Specification (BSC Issue 2);
 - 2) Windows Application Binary Interface WABI;
 - 3) MIT X Consortium Standards.

UNIX Berkeley Software Distribution (BSD) та System V Release 4 є результатом еволюції, уніфікації і стандартизації, що привели до кардинального відновлення системи. Ці потужні ОС розроблені в строгій відповідності з індустріальними стандартами, завдяки яким вони

відповідають вимогам з стійкості (portability), міжпроцесної взаємодії (interoperability), сумісності (compatibility) та масштабованості (scalability) – тобто можливості задовольняти вимоги як малих мікропроцесорних пристроїв, так і суперкомп'ютерів.

Також останнім часом активно розвиваються платформи для розвитку SMP (simmetnc multiprocessors) – тобто систем із багатопроцесорною паралельною архітектурою. У цьому напрямку існують прогресивні нароби для таких платформ як Hewlett Packard з HP 9000 Corporate Business Server T500 і OC HP/UX на базі System V; Motorola Series 900 M986 з OC UNIX SVR4; ICL Superserver (UNIX SVR4.2); Sun Microsystems UltraSparc Enterprise з OC Solaris.

Останнім часом характер світових економічних процесів і пов'язаних з ними комп'ютерних технологій визначається декількома найважливішими факторами:

- глобалізацією ринків і конкуренції;
- поширенням комп'ютерних і мережевих технологій;
- зменшенням кількості рівнів в ієрархічних управлінських структурах;
- вибухоподібним розширенням Internet та Intranet;
- становленням інформаційної економіки, коли ефективний розподіл прогресивних технологій надає змогу одержати переваги в конкурентній боротьбі.

В основі всіх цих тенденцій лежить один домінуючий фактор – безупинні зміни.

Глобалізація ринків і децентралізація корпорацій вимагають засобів віддаленої взаємодії і доступу до мережової інформації. Це повинно гарантуватися безперервністю інформаційних потоків як в середині організації, так і між компаніями, постачальниками і замовниками.

Однією з найважливіших задач, що постають перед компаніями в умовах зростаючої конкуренції – ефективне використання стратегічних комп'ютерних інформаційних ресурсів.

Сучасні крупні корпоративні мережі повинні забезпечити узгоджену роботу таких компонентів:

- успадкованих систем, що базуються на мейнфреймах;
- велику кількість персональних комп'ютерів і локальних мереж на основі персональних комп'ютерів та апаратних пристроїв;
- серверів для невеликих робочих груп;
- мережевих комп'ютерів з обмеженим набором локальних ресурсів, що базуються у своїй роботі на сервер;
- засобів зв'язку з постачальниками і замовниками через протоколи Internet.

Серед багатьох мережевих операційних систем, що претендують на роль інтегратора сучасного складного інформаційного світу (це в першу чергу різні варіанти OC UNIX і Microsoft NT), вважається BSD та Solaris. Вони надають користувачу повний набір необхідних сервісів.

ОС Solaris основана на промисловому стандарті UNIX System V Release 4. Вона забезпечує підтримку високоефективних додатків, побудованих в архітектурі "клієнт-сервер" і таких що функціонують у розподіленому мережному оточенні. Також BSD надає достатні ресурси невеликим робочим групам і підтримує Web-технологію, що необхідна для електронної комерції. Ці ОС відповідають практично всім специфікаціям і гарантують необмежений прозорий доступ до систем, серверів, принтерів, віддалених баз даних та інших ресурсів з масштабістю, достатньою для підтримки будь-яких додатків і конфігурацій.

Серед переваг BSD та Solaris відзначаються такі.

Простота використання та адміністрування

Середовище ОС надає користувачу потужні засоби віддаленого керування. Адміністратори можуть усувати несправності у мережі не заважаючи роботі користувачів. Мережі адмініструються в оперативному режимі. Додатки можна виконувати віддалено, з будь-якого робочого місця мережі. Середовище ОС дозволяє централізувати адміністрування і зменшити загальну вартість користування інформаційною системою.

Стрімке зростання Internet вимагає розробки інструментарію для HTML, C, C++ і Java, які функціонують на будь-якому вузлі мережі. Починаючи з версії 2.6 до складу Solaris входять Web-засоби. Два нових додатки, Solaris Web Start і Answerbook2, роблять використання ОС Solaris ще простішим. Інтерфейс користувача ОС оснований на новій версії системи CDE (Common Desktop Environment) і містить у собі нове керуюче програмне забезпечення.

Простота встановлення.

В ОС вперше реалізована можливість установки BSD та Solaris за допомогою навігатора. Web Start – це Java-додаток, що робить швидке встановлення ОС і прикладних додатків. Стандартна конфігурація ОС встановлюється за допомогою декількох етапів. Для обліку специфічних вимог замовників є численні варіанти вибору серед стандартних конфігурацій технічних платформ.

Web Start, разом з апаратними можливостями SPARC-комп'ютерів, дозволяє встановлювати ОС по мережі - у режимі "клієнт - сервер", навіть якщо на цільовій машині взагалі не було ОС. Технологія Sun JumpStart спрощує масове встановлення на бездискових станціях і клієнтах.

Потужне інструментальне середовище

ОС надає розроблювачам документацію, програмні бібліотеки, засоби підвищення продуктивності, приклади програм і інструменти для тестування. У даному розділі розглядаються нові можливості, що допомагають працювати з великими файлами (які мають розмір більш 2

Гбайт), багатопотоковими додатками, X – серверами і бібліотеками. Серед потужних інструментальних засобів середовища можна виділити такі:

- Файлова система /proc перестала мати один каталог. В ній з'явилися підкаталоги, що містять інформацію про стан процесів, керуючі функції, такі, як точки спостереження (Watchpoints);
- Федеративні сервіси іменування (Federated Naming Services, FNS) приведені у відповідність зі специфікаціями X/Open - XFN CAE;
- Нова область локалізації для розроблювачів en_US.UTF-8 допускає одночасну підтримку введення/виведення кількома мовами;
- засоби створення графічного інтерфейсу користувача середовища, що використовує прикладний програмний рівень – CDE;
- командна мова Desktop KomShell (dtksh);
- демонстраційна програма (з вихідними текстами), що використовує можливості Motif, ToolTalk і інші;
- Інструментарій розроблювача (Solaris Developer Kit, SDK) тепер вбудований у середовище ОС і його не потрібно встановлювати окремо. ОС містить всю інформацію, необхідну для розробки додатків і графічних елементів;
- Засоби конфігурування графічного X-сервера дозволяють змінювати параметри пристроїв платформи-незалежним чином.

Адміністрування мереж і систем

ОС BSD допомагає сформувати стабільне і надійне мережеве оточення. Важливі засоби адміністрування мереж і систем дозволяють ОС BSD ефективно використовувати надані ресурси:

- підтримка протоколу NTP (Network Time Protocol);
- реалізація рішень Solstice Enterprise Agents (SEA);
- підтримка протоколу DHCP (Dynamic Host Configuration Protocol);
- нейтралізація відмов NFS засобами клієнтів;
- змінна довжина маски підмережі (на один номер мережі може застосовуватись кілька масок, при маршрутизації IP-пакета ОС BSD автоматично використовує найдовшу з придатних масок);
- маршрутні сокети (повна відповідність стандарту, встановленому в BSD 4.4);
- набори процесорів можуть бути передані в монопольне використання додатком чи групою додатків (команда /usr/sb/ln/psrset);
- багатопотоковість демона автомонтування дозволяє одночасно обслуговувати кілька запитів на монтування;
- простір імен NIS+ може швидко копіюватися і відновлюватися засобами NIS+ Backup і Past Restore;
- визначення серверів NIS+ дозволяє задавати різний порядок для різних клієнтів, регулюючи тим самим навантаження на сервери;
- підтримка NIS-сервера;

- використання дискового кешу CacheFS прискорює завантаження систем AutoCient і зменшує навантаження на мережу;
- використання інструментів встановлення та вилучення програмного коду та асоційованої документації (команди patchadd і patchnn);
- набір утиліт Isalist для SPARC-систем дозволяє виявити набори підтримуваних команд та вибрати найефективніші з них.

Засоби розробки

Середовище розробки ОС BSD надає найширший у промисловості вибір інструментів для підтримки всього життєвого циклу (розробка, розгортання, супровід) складних розподілених продуктів:

1. Workshop

Продукти Workshop підтримуються ОС BSD для платформ SPARC і x86. Стратегія багатоплатформності дозволяє розроблювачам створювати додатки для різних платформ, не вивчаючи безліч різнотипних інструментів з декількох середовищ. Завдяки повній відповідності специфікаціям SVR4 Application Binary Interface (ABI) вихідний код підтримується на багатьох платформах.

2. Visual Workshop

Лінія продуктів Visual Workshop являє собою передову розробку в виробництві UNIX-середовище. Workshop включає повний набір інструментальних засобів для найпопулярніших мов, серед яких C, C++, Ада, Фортран. Багатопотокові компілятори та налагоджувачі дозволяють легко створювати додатки, що використовують переваги багатопроцесорних робочих станцій і серверів. ProWorks - це набір інструментів, що підвищують індивідуальну продуктивність праці розробника, а ProWorks/TeamWare містить інструменти керування кодом, версіями, засоби паралельного виконання "make", що дозволяють групам розробників координувати зусилля під час роботи над великими проектами. ProWorks/iMPact містить засіб побудови багатопотокових додатків.

У пакет Workshop для кожної мови входять відповідні компілятори плюс індивідуальний комплект ProWorks, ProWorks/TeamWare і proWorks/iMPact.

Visual Workshop C++ - це інтегрований комплект інструментів візуальної розробки, що підвищує продуктивність праці програмістів. У нього входять:

- візуальний дизайнер інтерфейсів;
- інтегроване середовище програмування, що допомагає переглядати, редагувати, компілювати та відлагоджувати додатки;
- мовні підсистеми C і C++;
- засоби розробки багатопотокових багатопроцесорних додатків;
- засоби конфігураційного керування параметрами мереж.

3. Workshop OpenStep

Workshop OpenStep надає розробникам комплект передових графічних засобів для створення потужних об'єктно-орієнтованих додатків. Об'єктні технології дозволяють вирішувати багато ключових проблем сучасного програмування, у тому числі:

- зменшення часу і вартості розробок;
- підвищення якості програмного забезпечення;
- багаторазове використання програм;
- спрощення доступу до сервісів;
- підвищення рівня сумісності;
- створення масштабних корпоративних інформаційних систем, які є інваріантними щодо особливостей мережі.

4. Internet Workshop

Internet Workshop – це інтегрований інструментарій, що містить все необхідне для створення систем Internet/Intranet. Розробнику надається можливість будувати додатки в тривірневій архітектурі "клієнт-сервер" з тонкими клієнтами на основі Java. У Internet Workshop входять 3 основних компоненти:

- Visual Workshop C++ – засіб розробки надійних, ефективних серверних систем;
- NEOWorks – засіб інкапсуляції і поширення об'єктів, побудованих за допомогою Visual Workshop, по корпоративній мережі, що дозволяє підвищити гнучкість системи і сприяє багаторазовому використанню програмного забезпечення;
- Java Workshop – засіб створення клієнтських додатків у Web-оточенні і поширення їх на всі настільні системи.

5. Java Workshop

За допомогою Java Workshop розроблювачі можуть створювати Java-аплети, що додають Web-сторінкам динамічні, інтерактивні риси. Найчастіше для цього не потрібно писати ні одного рядка коду, тому що можна скористатися зразками апаратів, які існують у Java Workshop. Ще одна можливість – це створення додатків, що можуть виконуватися на багатьох платформах де є віртуальна Java-машина.

Java Workshop містить все необхідне для створення і публікації мобільних Java-додатків:

- навігаційний Web-інтерфейс;
- інтегрований інструментарій для швидкої та ефективної розробки;
- засоби Web-публікації для організації Java-проектів та їх впровадження у Web;
- багатоплатформність, що дозволяє вести розробки в середовищах Unix і Microsoft Windows.

Мережеві засоби

Програмне забезпечення WebNFS дозволяє здійснювати доступ до файлових систем через Web по протоколу NFS. Цей протокол є достатньо надійним і забезпечує велику пропускну здатність при високій

завантаженості серверів. Крім того, файли можна зробити загальнодоступними без накладних витрат, пов'язаних з організацією анонімних ftp-послуг.

Мережні засоби ОС основані на технології ONC+ (Open Network Computing). ONC+ – це побудований на базі TCP/IP набір сервісів, інструментів і прикладних програмних інтерфейсів, включаючи мережеву файлову систему NFS (Network File System). Серед можливостей наданих користувачам ONC+, з'явилися:

- глобальний поділ файлів засобами NFS;
- високоефективне спільне використання принтерів;
- централізоване адміністративне сховище;
- захищені поширені мережеві сервіси каталогів та файлів;
- поліскранний графічний і текстовий віддалений доступ до протоколів Internet;
- доступ до віддалених систем і обмін даними в різних середовищах.

Мережеві об'єкти

Мережева об'єктна технологія дозволяє здійснювати доступ з додатків до об'єктів і їх розподілених сервісів у масштабах корпоративної мережі незалежно від місцезнаходження додатків і об'єктів. Індивідуальний об'єкт можна змінити без ризику "зіпсувати" додаток або бізнес-процес, що він моделює. Якщо оновлений об'єкт виявиться непридатним, то легко відновити попередню версію прикладної програми. Таким чином підтримується плавна, покрокова еволюція складних систем. У сполученні з мережевими засобами, об'єктна технологія породжує новий клас корпоративних додатків – гнучких, потужних, адміністративно централізованих, які не потребують великих витрат на створення.

Наявність підтримки багатьох платформ

Програмне забезпечення ОС масштабується для широкого спектра системних конфігурацій, від ізольованої робочої станції до корпоративного інформаційного середовища. ОС працює на платформах SPARC і x86. На всіх цих платформах, включаючи симетричні багатопроцесорні (Symmetrical Multi-Processing, SMP), забезпечується однакова функціональність. Це дозволяє підібрати для кожної конкретної задачі відповідну платформу, а за необхідності завжди можливий перехід до потужнішої платформи, що буде вимагати мінімальних зусиль. ОС основана на єдиних вихідних текстах. Тим самим гарантується, що наступні версії будуть мати ті ж можливості, функціональність і прикладні програмні інтерфейси.

Замовник може вибрати недорогу платформу і при цьому мати всі переваги стійкої, безпечної ОС комерційного класу з розвинутою функціональністю і кращими мережними можливостями. Персональні

комп'ютери можуть використовуватися як повноправні вузли корпоративного інформаційного середовища. Завдяки багатоплатформності користувачі мають можливість вибрати апаратну основу, що відповідає їх поточним потребам, будучи впевненими, що вони зможуть зберегти свої інвестиції і тоді, коли їхні потреби зростуть або зміняться.

Жодна ОС не охоплює з подібною ефективністю найбільш популярні RISC і CISC архітектури, тому BSD та Solaris по праву можна охарактеризувати як кращі довгострокові основи для корпоративних мереж та інтеграції різних апаратних платформ.

Безпечні методи роботи

ОС BSD забезпечує ефективну безпеку в масштабах різномірної мережі. ОС BSD підтримує різні моделі безпеки, що дозволяє гнучко будувати поточні і перспективні запити замовників. Перерахуємо деякі нові можливості ОС BSD по контролю за доступом, шифруванням та аутентифікацією:

- NFS Kerberos 5 - засіб аутентифікації користувачів у мережі;
- RPCSEC_GSS - модифікована реалізація віддаленого виклику процедур, що підтримує нові, більш потужні засоби забезпечення аутентифікації, конфіденційності і цілісності. Реалізація відповідає специфікаціям GSS-API;
- архітектура зі змінними модулями аутентифікації дозволяє побудувати нові механізми аутентифікації без зміни таких команд, як login, ftp чи telnet;
- використана поширена реалізація DNS-сервісу – Berkeley Internet Name Daemon (BIND).

Засоби безпеки ОС BSD забезпечують такі типи захисту:

- контроль входу в систему служить для перевірки привілеїв користувачів і дозволяє захиститися від "маскарадингу" (замаскованих процесів користувачів);
- керування доступом до системних ресурсів і підзвітність дій користувачів. Це засоби, що служать для розмежування доступу до ресурсів і даних у межах системи та мережі. У цій області частіше відбуваються випадкові порушення безпеки;
- сервіси, додатки та утиліти безпеки на рівні взаємодії "клієнт-сервер" дозволяють захиститися від прослуховування мережі. Цей тип захисту включає базові засоби керування доступом до сервісів типу rlogin, ssh, telnet і інших віддалено-доступних сервісів;
- керування мережевим доступом захищає від вторгнень із зовнішнього середовища, наприклад через Internet. Цей тип захисту реалізують додаткові продукти, такі як FireWall та багато інших.

Засоби безпеки ОС BSD дають користувачам прозорий доступ до ресурсів усієї компанії, а адміністраторам – необхідні інструменти для захисту мережі від порушень безпеки. Удосконалення контролю доступу, сумісні зі світовими специфікаціями з інформаційної безпеки, роблять

систему BSD більш стійкою до вторгнень неавторизованих користувачів і дозволяють клієнтам реалізувати більш витончену політику керування паролями з застосуванням таких засобів, як шифрування, контроль термінів дії, автоматичне закінчення терміну дії для файлів і системних ресурсів.

Java

В ОС середовища входять віртуальна Java-машина і навігатор HotJava. Останній можна рекомендувати для перегляду не тільки WWW, але і для роботи з документацією від Sun.

Віртуальна Java-машина

Java Virtual Machine 1.1 утворить Java-платформу для ОС BSD та Solaris. Вона сумісна з інструментальним середовищем Java Development Kit (JDK) версії 1.1 та вище компанії JavaSoft. До складу віртуальної машини входять стандартний Java-компілятор (javac), інтерпретатор (Java), дизасемблер класів (javap), а також генератор файлів, що включаються, мовою C і спільних файлів (javab).

Графічні засоби

Графіка була і залишається невід'ємною частиною середовища BSD, де користувачі можуть безпосередньо застосовувати безліч готових графічних додатків.

Стрімкий розвиток мультимедійних додатків призвело до зростання ролі засобів графіки та обробки зображень. Новаторська графічна архітектура в поєднанні з мережевими можливостями BSD допомагає розроблювачам збагатити графікою корпоративне інформаційне середовище, застосувати розвинуті графічні технології у важливих додатках і системах електронної комерції.

BSD надає користувачам ряд могутніх графічних бібліотек, серед яких XIL, XGL, OpenGL, Xlib, PostScript, Display PostScript і Kodak Color Management System (KCMS).

XIL Imaging Library - це багатопотокова бібліотека, що містить інтерфейс обробки зображень і підтримки цифрового відео. Вона придатна для додатків обробки документів, попереднього кольорового друку, генерації і відтворення цифрового відео. XIL Imaging Library має повний набір функцій для маніпулювання зображеннями і їхнього відтворення, включаючи арифметичні і логічні операції, пікселеву геометрію, стиснення, керування кольорами, відображення. Вона істотно прискорює розробку комерційних і технічних додатків, пов'язаних з обробкою зображень.

XGL Graphics Library - бібліотека дво- і тривимірної геометрії для додатків, у яких вимагаються функції геометричних перетворень і відображення. Бібліотека підтримує мобільність для різних апаратних платформ і оптимально використовує можливості графічних

прискорювачів. Також забезпечується прозора підтримка накладень для прискореної обробки меню і монохромних перекриттів.

BSD цілком підтримує апаратні засоби потужних графічних прискорювачів компанії Sun Microsystems, включаючи системи TurboZX, SX і Creator Graphics. Крім того, BSD забезпечує прискорену роботу конвеєра для Artist Graphics XJ1000 і MediaVision PG1024.

Програмна оболонка PEX належить до відомої реалізації специфікацій MIT PEX, що є розширенням X Window для тривимірної графіки. PEX підтримує безпосередній режим для графічних відображень, переданих по мережі з використанням X-протоколу. PEX надає програмістам більш широкі можливості створення графічних додатків для різномірних мережових середовищ.

До kodak Color Management System (KCMS) належить велика кількість прикладних програмних інтерфейсів і бібліотек, що дозволяють описувати і контролювати кольорні функції широкого спектра пристроїв: моніторів, сканерів, принтерів, пристроїв запису на плівку. Ці засоби забезпечують надійну обробку кольорових документів при введенні і відтворенні на різномірних пристроях.

BSD утворює міцний фундамент, на якому можуть бути побудовані додатки, що використовують геометричні перетворення, обробку зображень і поліекранний режим, що маніпулюють двовимірними і тривимірними об'єктами. Продукт OpenGL, який поставляється окремо компанією Sun Microsystems, являє собою власну реалізацію промислового стандарту OpenGL. Він забезпечує могутні засоби і дозволяє розробляти багатоплатформні додатки з тривимірною графікою.

Наявність різних мов

В процесі інтеграції світової економіки усе необхіднішими стають засоби для налаштування програмного забезпечення на різне мовне і культурне оточення. Користувачі хочуть взаємодіяти з програмами на своїй рідній мові і застосовувати звичні угоди про позначення часу, грошових одиниць і т.п.

ОС BSD відповідає сформульованим вимогам. Вона цілком підтримує 5 європейських мов (французький, німецький, італійський, шведський і іспанський) і 4 азіатських (японський, спрощений китайський, традиційний китайський і корейський). У ОС BSD додана підтримка нових областей локалізації для Росії, Австрії, Чехії, Угорщини, Польщі, Латвії, Литви, Естонії, Греції і Туреччині. Підтримка наборів символів, фонтів, кодувань для нових областей забезпечується у всіх локалізованих версіях ОС BSD, включаючи англійську. Це істотно спрощує розробку мовних додатків.

Оскільки російська і грецька мови основані на нелатинських алфавітах, для повної підтримки відповідних областей локалізації потрібна спеціальна клавіатура.

У ОС BSD додані дві області локалізації, що відповідають стандарту ISO 10646 і специфікаціям Unicode 2.0. Це en_US.utf-8 і ko.utf-8. Вони вперше в середовищі підтримують багатомовне введення/виведення. ОС BSD дозволяє легко створювати прикладні системи на різних мовах. Завдяки інтернаціоналізації прикладних програмних інтерфейсів в ОС BSD створений розроблювачами бінарний код буде правильно виконуватися в будь-якому національному оточенні.

Наявність документації

Answerbook2 являє собою нове покоління документованих систем, оснований на Web-технології. За допомогою будь-якого навігатора (зокрема, нового навігатора HotJava) користувачі можуть переглядати і друкувати великий комплект документації, що поставляється з ОС. Розвинуті засоби пошуку дозволяють легко відшукати потрібні фрагменти.

Автоматичне розпізнавання устаткування

Нова утиліта, що є частиною нової версії завантажувана у редакції для платформ x86, допомагає сконфігурувати пристрій. Вона надійно визначає наявне устаткування, враховує ресурси, необхідні кожному апаратному компоненту, надає користувачу список завантажувальних пристроїв.

Інтерфейс користувача

В цьому пункті мається на увазі інтерфейс користувача, оснований на CDE, який також підтримується OpenWindows.

CDE – це розвинуте середовище, що забезпечує простий, однаковий інтерфейс для усіх версій ОС UNIX. Компанії Sun Microsystems, Hewlett-Packard, IBM і Novell спільно розробили стандарт користувацьких і прикладних інтерфейсів на основі X Window System і специфікацій Motif.

Глобальний сервіс каталогів NIS+

NIS+ – це захищене високоефективне розподілене сховище даних. Адміністративні дії, такі як додавання, вилучення, перепризначення систем або користувачів, легко виконуються за допомогою ефективних засобів зміни інформації, що зберігається в NIS+. NIS+ спрощує адміністрування малих мереж і легко масштабується для існуючих методів керування корпоративними мережами, що містять десятки тисяч систем та користувачів.

Сумісність з локальними мережами ПК

ОС підтримує стек протоколів IPX/SPX компанії Novell, забезпечуючи базову сумісність з локальними мережами NetWare. Успішна взаємодія з ними – властивість важлива з погляду організації

різнорідних мереж. Продукти Solaris Server і The Solaris Intranet Server Extension реалізують функціональність Novell NetWare (IPX/SPX), включаючи інтеграцію в існуючі локальні мережі NetWare. Вони надають користувачу відмінну можливість перейти від NetWare-серверів до розподілених засобів існуючих потужностей ОС Unix.

Федеративні сервіси

Федеративні сервіси (Federated Services) ОС – це інфраструктура і безліч інтерфейсів, які ховають від користувачів і розроблювачів додатків мережеві інтерфейси нижнього рівня. Таким чином, програмісти і користувачі звільняються від необхідності вивчати деталі різних мережних сервісів. Це приводить до зниження витрат і спрощення побудови додатків, що функціонують у різнорідних середовищах. До ОС входить X/Open Federated Naming (XFN) - корпоративний сервіс каталогів та іменувань, що забезпечує доступ до різних середовищ (DCE, ONC+, DNS) і їхню інтеграцію.

Найважливішою перевагою використання ОС є застосування в ній відкритих стандартів. Відкриті системи і стандарти завжди займали важливе місце в діяльності різних компаній. Стандарти були і залишаються фундаментом ОС Unix, що відіграє провідну роль у розробці багатьох домінуючих сьогодні специфікацій: POSIX, X/Open, OSF і OMG.

У число відомих стандартів, що підтримує ОС, входять:

- стандарти інтерфейсів: X/Open, X/Open UNIX 95 (Spec 1170), X/Open XFN CAE, X/Open XPG3, XPG4 і XPG4.2, Federated Naming (XFN); IEEE POSIX. 1, .1b, .13 і POSIX.2 (Portable OS Interface), POSIX 1003.1b;
- стандарти керування енергоспоживанням: EPA EnergyStar;
- графічні стандарти: X11, PEX, PostScript, Display PostScript OpenGL (за допомогою продуктів третіх фірм);
- стандарти настільного середовища: CDE (Common Desktop Environment);
- стандарти мережної взаємодії: ONC+, NetWare IPX/SPX, додатково - SNA, DECnet і інші;
- об'єктні стандарти: OMG CORBA і асоційовані специфікації.

Сімейство продуктів Workshop NEO

Сімейство продуктів Workshop NEO реалізує описаний вище новий підхід до побудови програмних систем корпоративного масштабу. Оскільки об'єктне програмування підтримує багаторазове використання програмних компонентів, то створення складних додатків вимагає значно менше зусиль та дозволяє істотно прискорити побудову і спростити процес еволюції додатків у тих компаніях, де використовується мережева об'єктна технологія.

Продукти Workshop NEO базуються на специфікаціях CORBA (Common Object Request Broker Architecture), вироблених групою OMG (Object Management Group). Workshop NEO дозволяє привнести технологію "клієнт-сервер" в Internet і розгорнути корпоративну мережу сервісів з використанням ефективних масштабних мережесих об'єктів.

В загальному випадку, Workshop NEO містить:

- Joe 2.0 - засіб об'єднання Java і Workshop NEO 2.0 у середовище, яка пристосована для створення надійних масштабних інтерактивних Web-додатків, що опираються на існуючу корпоративну мережеву інфраструктуру;
- NEO Connectivity for Microsoft Windows Desktops – рішення Sun для тісної інтеграції додатків, що працюють на настільних системах під керуванням Microsoft Windows та NEO-систем, що функціонують під керуванням ОС Solaris на корпоративних серверах. Подібне рішення щонайкраще підходить для організацій, що бажають будувати Internet-додатки в архітектурі "клієнт-сервер" з використанням Workshop NEO і зі збереженням інвестицій у настільні системи під Microsoft Windows;
- Solstice NEO – набір інструментів адміністрування, що поставляється з Workshop NEO і дозволяє керувати мережею об'єктів з будь-якого вузла, здійснюючи балансування навантаження.

Таким чином, узагальнивши сказане, можна стверджувати, що ОС BSD та Solaris є найбільш прийнятним варіантом для корпоративних та глобальних комп'ютерних мереж. Всі вони мають багатопотокове ядро з підтримкою багатопроцесорності; відповідають стандартам для відкритих систем, забезпечуючи сумісність з іншими оточеннями; містять інструменти розробки та адміністрування світового класу; підтримують багато відомих додатків третіх фірм; надають користувачам середовище для виконання додатків MS Windows. Масштабність BSD та Solaris сягає від робочих станцій до суперкомп'ютерів, що дозволяє замовникам при потребі створювати будь-які різноманітні корпоративні мережі з нарощуваною потужністю.

Контрольні питання

1. Назвіть відомі прийняті стандарти і рекомендації для ОС UNIX.
2. Назвіть компоненти, які повинні забезпечити узгоджену роботу сучасних корпоративних мереж.
3. Охарактеризуйте загальну властивість мережесих ОС FreeBSD та Solaris з огляду на можливості використання та адміністрування інформаційних ресурсів.

4. Охарактеризуйте загальну властивість мережевих ОС FreeBSD та Solaris з огляду на можливості використання інструментального середовища.

5. Охарактеризуйте загальну властивість мережевих ОС FreeBSD та Solaris з огляду на можливості адміністрування мереж і систем.

6. Охарактеризуйте загальну властивість мережевих ОС FreeBSD та Solaris з огляду на можливості використання потужних засобів розробок.

7. Охарактеризуйте загальну властивість ОС FreeBSD та Solaris з огляду на можливості використання потужних мережевих засобів та об'єктів.

8. Охарактеризуйте загальну властивість мережевих ОС FreeBSD та Solaris з огляду на можливості підтримки багатьох платформ та використання засобів безпеки.

9. Охарактеризуйте загальну властивість мережевих ОС FreeBSD та Solaris з огляду на можливості використання графічних засобів та інтерфейсу користувача.

10. Охарактеризуйте загальну властивість мережевих ОС FreeBSD та Solaris з огляду на можливості роботи з різними мовами та використання програмних продуктів Workshop NEO.

Література

1. Хошаба О.М. Протоколи основних служб міжнародної комп'ютерної мережі Internet. Лабораторний практикум. Навчальний посібник - В.: ВДТУ, 2000. -104 с.
2. Хошаба О.М. Методичні вказівки до виконання лабораторних робіт з дисципліни "Мікропроцесори та програмне забезпечення інтелектуальних систем" Частина І. Операційна система Unix. Для студентів бакалаврських напрямків 6.0915 та 6.0804. -Вінниця: ВДТУ, 1998. - 42 с.
3. Хошаба О.М. Ефективна та безпечна робота в міжнародній комп'ютерній мережі Internet. Лабораторний практикум. Навчальний посібник. -Вінниця: ВДТУ, 1999. - 60 с.
4. Дунаев С. Unix-сервер. В 2-х томах. Том 1.-М.: "Диалог-МИФИ", 1998.- 304с.
5. Ларри Шамер, Крис Негус Unix без проблем. Руководство пользователя. -М.: "Бином", 1996.-352с.
6. Болски М.И. Язык программирования Си. Справочник. -М.: Радио и связь, 1988. – 96с.

Навчальне видання

Олександр Мирославович Хошаба

ОПЕРАЦІЙНІ СИСТЕМИ КОМП'ЮТЕРНИХ МЕРЕЖ
Навчальний посібник

Оригінал-макет підготовлено автором

Редактор С.А.Малішевська

Навчально-методичний відділ ВНТУ
Свідоцтво Держкомінформу України
серія ДК №746 від 25.12.2001
21021, м. Вінниця, Хмельницьке шосе, 95, ВНТУ

Підписано до друку 26.08.04 р. Гарнітура Times New Roman

Формат 29,7x42 $\frac{1}{4}$

Папір офсетний

Друк різнографічний

Ум. друк. арк. 6.29

Тираж 75 прим.

Зам. № 2004-135

Віддруковано в комп'ютерному інформаційно-видавничому центрі

Вінницького національного технічного університету

Свідоцтво Держкомінформу України

серія ДК №746 від 25.12.2001

21021, м. Вінниця, Хмельницьке шосе, 95, ВНТУ