МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М.Є.Жуковського
“Харківський авіаційний інститут”

О.І. Риженко, В.В. Шевель

ОСНОВИ АРХІТЕКТУРИ ОС UNIX: РОБОТА В СЕРЕДОВИЩІ КОМАНДНОГО ІНТЕРПРЕТАТОРА bash

Навчальний посібник

Харків “ХАІ” 2009

УДК 004.451(075.8)+681.3.06

Риженко О.І. Основи архітектури ОС UNIX: робота в середовищі командного інтерпретатора bash: навч. посібник / О.І. Риженко, В.В. Шевель. – Х.: Нац. аерокосм. ун-т “Харк. авіац. ін-т”, 2009. – 82 с.

Викладено основи архітектури ядра операційної системи UNIX і середовища користувача для програмування засобами командного інтерпретатора bash. Коротко розглянуто окремі компоненти ОС UNIX: файлова підсистема та підсистема керування процесами. Наведено значну кількість прикладів, що сприяють засвоєнню матеріалу.

Для студентів спеціальностей “Інформатика і обчислювальна техніка”, “Прикладна математика та інформатика”, “Обчислювальна техніка і програмування”, а також для бакалаврів і спеціалістів, магістрів і аспірантів, які спеціалізуються в області комп’ютерних технологій.

Іл. 15. Табл. 11. Бібліогр.: 9 назв

Рецензенти: канд. техн. наук, доц. С.В. Красников,
канд. техн. наук В.О. Коваль

Зміст

1. ОГЛЯД ОСОБЛИВОСТЕЙ ОС UNIX

1.1. Історія створення і розвитку ОС UNIX

Операційна система (ОС) UNIX створена в Bell Telephone Laboratories (тепер AT&T Bell Laboratories) і зв'язана з іменами Кена Томпсона, Денніса Рітчі й Брайана Кернігана.

У 1969 році Кен Томпсон створив комп'ютерну гру "Space Travel", орієнтовану на 8-розрядний комп'ютер PDP-7 з 8К оперативної пам'яті та якісним графічним дисплеєм. На цьому комп'ютері була встановлена спеціально розроблена файлова система, яка підтримувала використання системи двома користувачами в режимі розподілу часу. Б. Керніган запропонував назвати цю систему UNICS (Uniplexed Information and Computing System), потім назва UNICS перетворилася в UNIX (вимовляється так само, але на одну літеру коротше).

Однак можливості PDP-7 не задовольняли потреби Bell Telephone Laboratories. У 1971 році UNIX була перенесена на новий 16-розрядний комп'ютер фірми Digital Equipment PDP-11/20. Документація цього варіанта системи, що була написана мовою асемблера, була опублікована у листопаді 1971 року й одержала назву *першої редакції ОС UNIX*.

Друга редакція ОС UNIX (1972 р.) була переписана мовою програмування B, яку розробив К. Томпсон; це була перша операційна система, написана не мовою асемблера. Однак сама мова B була досить примітивною.

У 1973 році Д. Рітчі розробив мову програмування C, яка є удосконаленою мовою програмування B. Томпсон і Рітчі переписали систему UNIX мовою C; це була *четверта редакція ОС UNIX*.

У 1974 році компанія Bell Labs оголосила про можливість безкоштовного одержання вихідних текстів UNIX для використання з метою навчання. *П'яту редакцію ОС UNIX*, що з'явилася у цей час, одержали Каліфорнійський університет (м. Берклі) й університет Нового Південного Уельсу (м. Сідней – Австралія).

У 1975 році компанія Bell Labs видала *шосту редакцію ОС UNIX*, відому як V6 чи дослідницький UNIX. Ця версія була першою комерційною системою, доступною поза Bell Labs. Велика частина системи була написана мовою C, що забезпечувало ОС UNIX V6 нову якість реально переносної операційної системи.

У 1976 році в університеті м. Берклі, де було встановлено UNIX V6 на комп'ютері PDP-11/70, Білл Джой заснував компанію BSD (Berkeley Software Distribution), а згодом він заснував і очолив компа-

нію Sun Microsystems. Б. Джой зібрав разом великий обсяг програмного забезпечення, названий Berkeley Software Distribution (BSD 1.0), і повний набір текстів, що включав UNIX V6, компілятор мови Паскаль, редактор ex (згодом редактор vi) та інші програми.

У 1977 році К. Томпсон і Д. Рітчі зробили повне перенесення ОС UNIX на новий комп'ютер з 32-розрядною архітектурою. Для цього Рітчі розширив і вдосконалив мову С і разом з Томпсоном цілком переписав підсистему керування оперативною й віртуальною пам'яттю і змінив інтерфейс драйверів зовнішніх пристроїв, щоб полегшити перенесення системи. Результатом роботи стала *сьома редакція* UNIX (UNIX Version 7). До складу нової версії системи увійшли: компілятор нового діалекту мови С PCC (Portable C-Compiler), новий командний інтерпретатор sh, названий на честь свого творця Bourne-shell, набір нових драйверів пристроїв та інші системні програми.

Згодом значний інтерес до ОС UNIX почали виявляти комерційні компанії — виробники комп'ютерів і програмного забезпечення. Це пояснюється тим, що з розвитком технології електронних схем значно зменшилась вартість виробництва нових однокристальних процесорів. Тому наявність по-справжньому мобільної операційної системи, перенесення якої на нову апаратну платформу не потребує багато часу та коштів, дозволяла оснастити нові комп'ютери якісним базовим програмним забезпеченням. У результаті до кінця 70-х років UNIX-подібні операційні системи були доступні на комп'ютерах з мікропроцесорами Zilog, Intel, Motorola і т.д. З'явилися тисячі комп'ютерів з ОС UNIX.

У 1982 році AT&T передала за межі Bell Labs свій перший варіант UNIX, що одержав назву UNIX System III. У цій системі знайшли втілення кращі якості UNIX Version 7, V/32 й інших варіантів UNIX, які використовувались у Bell Labs.

Нині не існує стандартної ОС UNIX, замість цього є безліч операційних систем, що мають власні назви й особливості. Найбільше застосування мають такі різновиди системи:

- *System V UNIX фірми AT&T*, у якій реалізоване заміщення сторінок і копіювання при запису, надано систему міжпроцесорної взаємодії IPC (InterProcess Communication) з розподілом пам’яті, чергою повідомлень і семафорами;

- *BSD UNIX* компанії Berkeley Sofftware Distribution – перша версія UNIX, перенесена на комп'ютери VAX. Останні версії, випущені у Берклі – 4.4 BSD і BSD Life – забезпечують могутню підтримку мережі і, на відміну від версій AT&T, широко доступні;

- *ОС OSF/1* – розроблена організацією OSF (Open SoftWare Foundation), створеною компаніями-виробниками обчислювальної

техніки (IBM, DEC, Hewlett-Packard) для розробки незалежної від AT&T версії системи;

- *версії UNIX, які використовують мікроядро* і забезпечують мінімізацію функцій, виконуваних ядром операційної системи, що спрощує налагодження та конфігурування системи (мікроядро MACH, розроблене в університеті Карнегі-Меллона, і мікроядро Chorus);

- *Linux* – вільно розповсюджувана версія UNIX, розроблена дослідником Хельсінського університету Л. Торвальдсом. Спочатку вона була розроблена для процесора Intel x386, зараз перенесена на інші апаратні платформи (наприклад, на сервери Alpha фірми DEC).

1.2. Стандартизація ОС UNIX

З появою на ринку комерційних версій ОС UNIX стала необхідною стандартизація системи, що полегшує перенесення програмного забезпечення і захищає користувачів і виробників.

У 1984 році був створений перший стандарт для програмного інтерфейсу UNIX. Його використовував комітет ANSI (American National Standards Institute) при опису бібліотек мови С.

У 1985 році був розроблений стандарт POSIX 1003.1-1988 (Portable Operating System Interface for Computing Environment), що визначив програмний інтерфейс програмного забезпечення (API – Application Programming Interface).

Іншими відомими стандартами, що відносяться до ОС UNIX, є:

- XPG3 (X/OPEN Portability Guide версії 3), що включає стандарт POSIX і стандарт на графічну систему X Windows System;

- SVID (System V Interface Definition), у якому визначалися вимоги до зовнішніх інтерфейсів UNIX версій System V;

- удосконалений стандарт X3.159-1989 мови програмування С, розроблений комітетом ANSI.

1.3. Причини популярності ОС UNIX

ОС UNIX, яка вперше з'явилася чотири десятиліття тому, нині використовується в усьому світі на комп'ютерах із широким діапазоном обчислювальних можливостей – від мікропроцесорів до великих ЕОМ. Популярність й успіх системи UNIX пояснюються декількома причинами:

- система написана мовою високого рівня С, завдяки чому її легко читати, розуміти, змінювати та переносити на інші комп'ютери (за оцінкою Д. Рітчі використання мови С на 20 - 40 % збільшило обсяг

системи і зменшило її оперативність порівняно з варіантом на асемблері);

- система має могутній модульний інтерфейс користувача та набір утиліт, кожна з яких вирішує свою спеціалізовану задачу, що дозволяє створювати з них складні програми;

- UNIX оснащена єдиною ієрархічною файловою системою, легкою в супроводі й ефективною в роботі; засобами файлової системи забезпечується не тільки доступ до даних, що зберігаються на диску, але і доступ до терміналів, принтерів, магнітних стрічок, мережі й навіть до оперативної пам'яті;

- система має простий інтерфейс для роботи з периферійними пристроями;

- система розрахована на широке коло користувачів, вирішує велику кількість задач із широким спектром послуг; кожен користувач може одночасно виконувати кілька процесів;

- є велика кількість вільно розповсюджуваних додатків, починаючи від найпростіших текстових редакторів і закінчуючи могутніми системами керування базами даних, що працюють під цією системою;

- система ховає від користувача архітектуру комп'ютера, що полегшує процес написання програм, які працюють на різних конфігураціях апаратних засобів;

- хоча операційна система та більшість команд написані мовою С, система підтримує ряд інших мов (Фортран, Бейсик, Паскаль, Ада, Кобол, Лісп, Пролог та ін.). Система UNIX може підтримувати будь-яку мову програмування, для якої є компілятор-інтерпретатор, і забезпечує системний інтерфейс, що встановлює відповідність між запитами користувача до операційної системи і набором запитів, прийнятих у UNIX.

1.4. Базові поняття ОС UNIX

Основними поняттями ОС UNIX є: користувач, привілейований користувач, інтерфейс користувача, програми, команди, процеси, потоки.

Користувач. Зареєстрований користувач системи – це особа, яка зареєстрована в облікових файлах системи і має облікове ім'я. Ядро ОС UNIX ідентифікує кожного користувача за його ідентифікатором UID (User Identifier), що присвоюється при реєстрації особи в системі. Крім того, кожен користувач відноситься до деякої групи, обумовленої ідентифікатором GID (Group IDentifier). Реєстрація нових користувачів виконується адміністратором системи. Користувач не може

змінити своє облікове ім'я, але може встановити і/чи змінити свій пароль. Паролі зберігаються в окремому файлі в закодованому вигляді. Усі користувачі ОС UNIX явно чи неявно працюють з файлами. Файлова система ОС UNIX має деревоподібну структуру. Проміжними вузлами дерева є каталоги з посиланнями на інші каталоги чи файли, а листи дерева відповідають файлам або порожнім каталогам. Кожному зареєстрованому користувачу відповідає деякий каталог файлової системи, що називається "домашнім" (home) каталогом користувача. При вході в систему користувач одержує необмежений доступ до свого домашнього каталогу та всіх каталогів і файлів, що містяться в ньому. Користувач може створювати, видаляти і модифікувати каталоги та файли, що містяться в домашньому каталозі. Доступ до всіх інших файлів визначається рівнем привілеїв користувача.

Інтерфейс користувача. Сучасні версії UNIX мають розвинуті графічні інтерфейси. Однак у будь-якій версії системи підтримується традиційний засіб взаємодії користувача із системою через інтерфейс командного рядка. Після входу користувача в систему для нього запускається один з командних інтерпретаторів. Загальна назва для будь-якого командного інтерпретатора ОС UNIX – shell (оболонка). Зазвичай в системі підтримується кілька командних інтерпретаторів зі схожими командними мовами. Командний інтерпретатор дозволяє користувачу вводити з клавіатури командний рядок, який може містити одну команду чи конвеєр (послідовність) команд. Після виконання чергового командного рядка shell видає запрошення на введення наступного командного рядка, і так доти, поки користувач не завершить свій сеанс роботи. Командні мови ОС UNIX можуть бути використані для написання складних програм на основі застосування механізму командних файлів (shell scripts), що можуть містити довільні послідовності командних рядків. При вказівці імені командного файла замість чергової команди інтерпретатор читає файл рядок за рядком і послідовно інтерпретує команди. Значення UID і GID для кожного зареєстрованого користувача зберігаються в облікових файлах системи і приписуються процесу, у якому виконується командний інтерпретатор, запущений при вході користувача в систему. Ці значення успадковуються кожним новим процесом, запущеним від імені користувача, і використовуються ядром системи для контролю прав доступу до файлів, виконання програм і т.д.

Привілейований користувач. Адміністратор системи, що теж є зареєстрованим користувачем, повинен мати більші можливості, ніж звичайні користувачі. В ОС UNIX ця задача вирішується шляхом виділення одного значення UID (нульового) суперкористувачу (superuser

чи root). Він має необмежені права на доступ до будь-якого файла і на виконання будь-якої програми, на нього не поширюються обмеження щодо використання ресурсів. Для звичайних користувачів установлюються такі обмеження: максимальний розмір файла, максимальне число сегментів поділюваної пам'яті, максимально припустимий простір на диску і т.д. Суперкористувач може змінювати ці обмеження для інших користувачів, але на нього вони не поширюються.

Програми. ОС UNIX одночасно є операційним середовищем використання існуючих прикладних програм і середовищем розробки нових додатків. Нові програми можуть писатися на різних мовах (Фортран, Паскаль, Модула, Ада й ін.). Однак стандартною мовою програмування в середовищі ОС UNIX є мова С (останнім часом С++).

Команди. Будь-яка командна мова сім'ї shell фактично складається з трьох частин: службових конструкцій, що дозволяють маніпулювати з текстовими рядками і будувати складні команди на основі простих команд; убудованих команд, виконуваних безпосередньо інтерпретатором командної мови; команд, що подаються окремими виконуваними файлами (системними утилітами і програмами користувача).

Процеси в ОС UNIX – це програми, що виконуються у власному віртуальному адресному просторі. Коли користувач входить у систему, автоматично створюється процес, що відповідає першій запущеній команді. Ця запущена програма, у свою чергу, може створити процес і запустити в ньому іншу програму і т.д.

Переадресування вхідних/вихідних даних. Зазвичай інтерактивні програми UNIX вводять текстові рядки з клавіатури термінала і виводять результуючі текстові рядки на екран. Механізм переадресування вхідних/вихідних даних забезпечує більш гнучке використання таких програм, дозволяючи здійснювати введення з файла вихідних даних інших програм і виведення у файл вхідних даних інших програм. Переадресування вхідних/вихідних даних ґрунтується на таких властивостях ОС UNIX:

- будь-яке введення/виведення трактується як введення з деякого файла і виведення у деякий файл (клавіатура і екран термінала теж інтерпретуються як файли);

- будь-якому файлу виділяються системою три значення дескрипторів: 1 - стандартне введення (**`stdin`**), 2 - стандартне виведення (**`stdout`**) і 3 - стандартне виведення діагностичних повідомлень (**`stderr`**);

- програма, що запущена в деякому процесі, успадковує від процесу, що її породив, усі дескриптори відкритих файлів.

У головному процесі інтерпретатора командної мови стандартним засобом введення є клавіатура термінала користувача, а стандартним засобом виведення результату та діагностичних повідомлень – екран термінала. Однак при запуску будь-якої команди можна повідомити інтерпретатору (засобами командної мови), який файл чи виведення якої програми мають виконувати функцію стандартного введення для програми, що запускається, і який файл чи введення якої програми мають виконувати функцію стандартного виведення чи виведення діагностичних повідомлень для програми, що запускається.

1.5. Функції ОС UNIX

Виконуючи різні елементарні операції з обслуговування запитів процесів користувача, ядро ОС UNIX забезпечує функціонування інтерфейсу користувача. Серед функцій операційної системи основними є:

- керування процесами (їх створення, завершення, припинення й організація взаємодії між ними);

- керування наданням часу центрального процесора (диспетчеризація); процеси працюють з центральним процесором у режимі розподілення часу: центральний процесор виконує процес, по завершенні відлічуваного ядром кванта часу процес припиняється і ядро активізує виконання іншого процесу; пізніше ядро запускає припинений процес;

- керування оперативною пам'яттю; ядро захищає адресний простір, виділений процесу, від утручання ззовні, дозволяючи процесам спільно використовувати ділянки адресного простору на визначених умовах; якщо системі потрібна вільна пам'ять, ядро тимчасово вивантажує весь процес (система зі своппінгом) чи окремі сторінки процесу (система з заміщенням сторінок) на зовнішні запам'ятовуючі пристрої;

- керування зовнішньою пам'яттю (виділення зовнішньої пам'яті під файли користувача, структуризація файлової системи, захист файлів від несанкціонованого доступу);

- керування доступом процесів до периферійних пристроїв (терміналів, стрічкових пристроїв, дисководів, мережного устаткування).

1.6. Апаратне середовище ОС UNIX

Процеси користувача у системі UNIX виконуються в двох режимах: режимі користувача і режимі ядра. Запущений процес користувача функціонує *в режимі користувача*. При звертанні процесу до операційної системи режим виконання процесу переключається з режиму користувача *на режим ядра*: операційна система обслуговує запит користувача (у випадку невдалого завершення операції процесу повертається код помилки) і виконання повертається в режим користувача**.** Навіть якщо користувач не звертається до системи з запитами, система виконує переключення в режим ядра для ведення облікових операцій, пов'язаних з процесом користувача, обробки переривань, планування процесів, керування розподілом пам'яті й т.д. Основні розходження між цими двома режимами подані в табл. 1.1.

Таблиця 1.1

Основні характеристики режимів виконання процесу

Режим користувача	*Режим ядра*
Процеси мають доступ тільки до своїх власних інструкцій і даних, але не до інструкцій і даних ядра або інших процесів	Процесам доступні адресні простори ядра й інших користувачів (за певних умов)

Незважаючи на те, що система функціонує в одному з двох режимів, ядро діє від імені процесу користувача. Ядро не є якоюсь особливою сукупністю процесів, що виконуються паралельно з процесами користувача; воно є складовою частиною будь-якого процесу користувача. Наприклад, командний процесор shell зчитує вхідний потік з термінала за допомогою запиту до операційної системи. Ядро операційної системи від імені процесора shell керує функціонуванням термінала і передає символи, що вводяться, процесору shell. Shell переходить у режим задачі, аналізує потік символів, уведених користувачем, і виконує задану послідовність дій, що можуть зажадати виконання й інших системних операцій.

1.6.1. Переривання й особливі ситуації

ОС UNIX дозволяє деяким пристроям (зовнішні пристрої введення-виведення, системний годинник) асинхронно переривати роботу центрального процесора. Переривання цього типу називаються апаратними зовнішніми перериваннями, оскільки викликаються поді-

ями, зовнішніми стосовно процесора. Після одержання сигналу переривання ядро операційної системи зберігає поточний контекст активного процесу, встановлює причину переривання й обробляє його. Після обробки переривання перерваний контекст відновлюється та виконання процесу продовжується.

Особливі ситуації (чи апаратні внутрішні переривання) пов'язані з виникненням незапланованих подій, що викликані процесом, таких, як неприпустима адресація, завдання привілейованих команд, поділ на нуль і т.д. Особливі ситуації виникають при виконанні команди; система, обробивши особливу ситуацію, намагається запустити команду знову. Для обробки переривань і особливих ситуацій у системі UNIX використовується однаковий механізм.

1.6.2. Рівні переривання процесора

Черговість обробки переривань від пристроїв визначається їх пріоритетами. У процесі обробки переривань від пристроїв ядро враховує пріоритети і блокує обслуговування переривання з низьким пріоритетом на час обробки переривання з вищим пріоритетом.

На рис. 1.1 показано рівні переривань ОС UNIX.

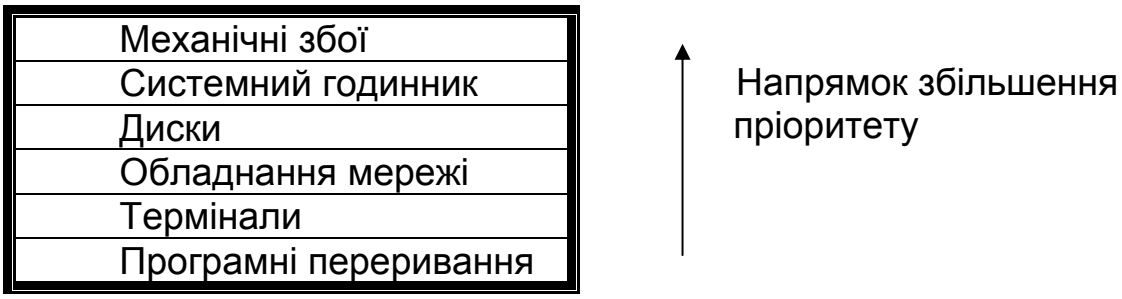


Рис. 1.1. Рівні переривань ОС UNIX

1.6.3. Розподіл пам'яті

Постійно в оперативній пам'яті розташовується ядро системи і процес, що виконується в цей момент, або його частина. У процесі компіляції програма-компілятор генерує послідовність віртуальних адрес виконуваної програми, що є адресами змінних, інформаційних структур і адресами інструкцій і функцій таким чином, якби комп'ютером виконувалася тільки одна ця програма. При запуску програми ядро виділяє для неї місце в оперативній пам'яті, при цьому збіг віртуальних адрес, генерованих компілятором, з фізичними адресами зовсім необов'язковий. Ядро, взаємодіючи з апаратними засобами,

транслює віртуальні адреси у фізичні, тобто відображає адреси, генеровані компілятором, у фізичні, машинні адреси.

1.7. Вступ до архітектури ОС UNIX

1.7.1. Дворівнева модель ОС UNIX

Архітектура ОС UNIX базується на дворівневій моделі системи, яку показано на рис.1.2.

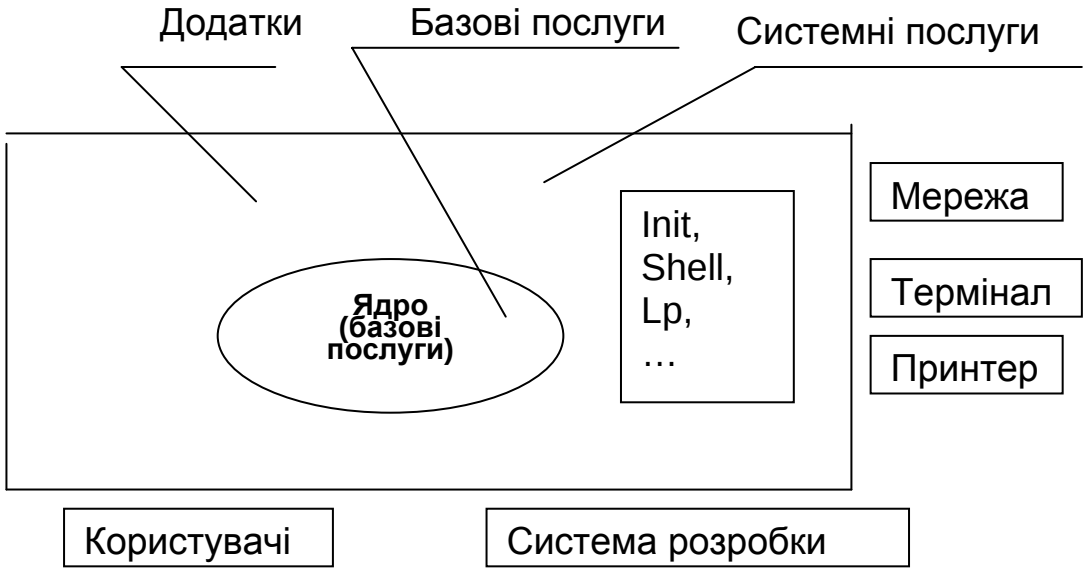


Рис. 1.2. Архітектура ОС UNIX

На першому рівні (у центрі) знаходиться ядро системи (kernel), що безпосередньо взаємодіє з апаратним забезпеченням комп'ютера, ізолюючи прикладні програми від особливостей його архітектури. Ядро надає прикладним програмам набір послуг (робота з файлами, процесами, міжпроцесорна взаємодія). Прикладні програми запитують послуги ядра за допомогою системних викликів.

Другий рівень складають додатки, як системні (що забезпечують функціональність системи), так і користувальницькі (які забезпечують розв’язання конкретних задач). Схеми взаємодії з ядром і системних, і користувальницьких додатків однакові.

1.7.2. Ядро ОС UNIX

Ядро ОС UNIX керує ресурсами комп'ютера і надає користувачам базовий набір послуг.

Внутрішню архітектуру ядра показано на рис. 1.3.

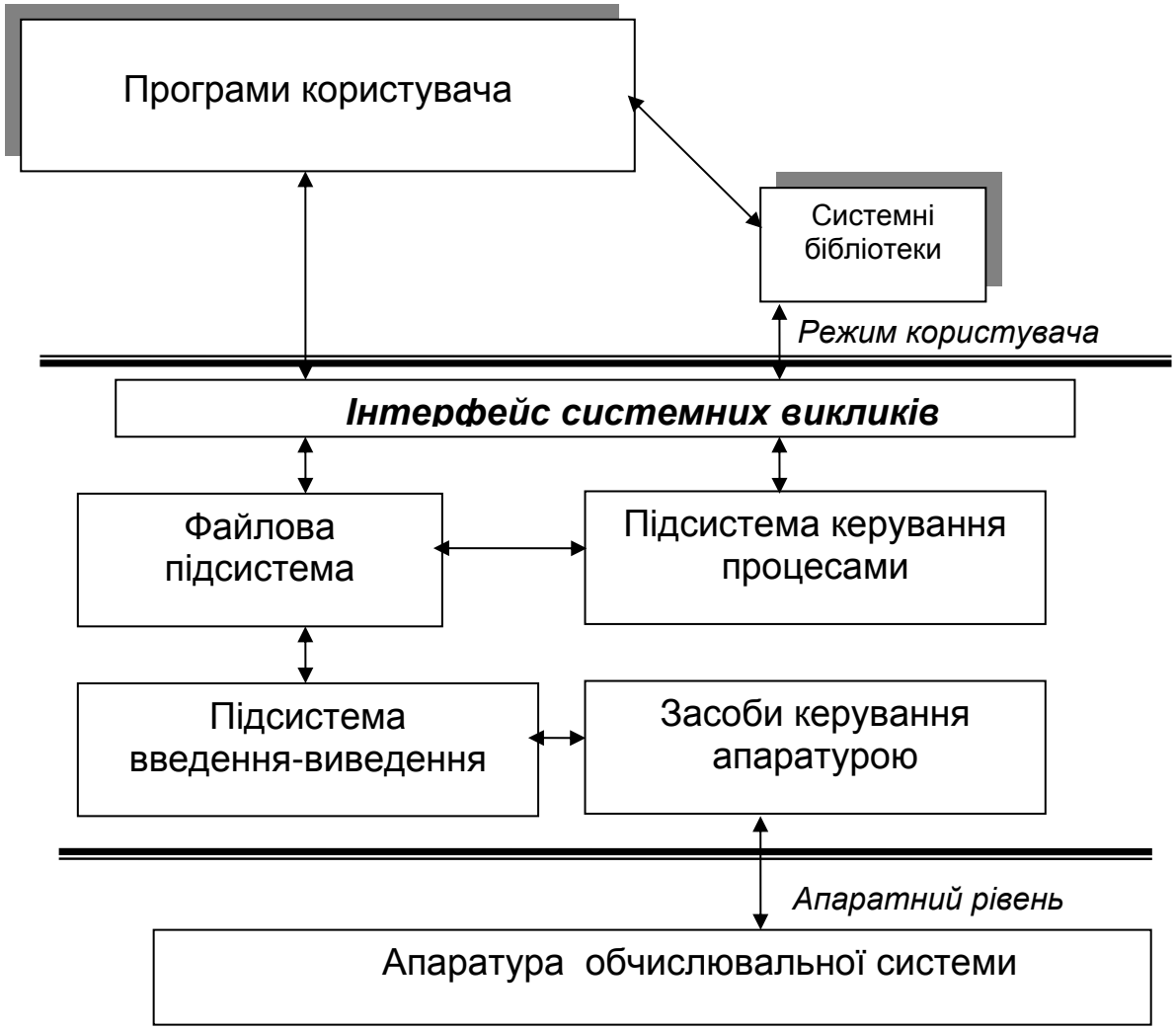


Рис. 1.3. Архітектура ядра ОС UNIX

Прикладні програми взаємодіють з ядром через інтерфейс системних викликів. Процес запитує послугу шляхом системного виклику визначеної процедури ядра, аналогічно виклику бібліотечної функції. Ядро від імені процесу виконує запит і повертає процесу необхідні дані. До основних функцій ядра ОС UNIX відносяться:

- ініціалізація системи: завантаження повного ядра в пам'ять комп'ютера і запуск ядра;

- керування процесами: створення, завершення, відстеження існуючих процесів, розподіл між запущеними процесами часу процесора (чи процесорів у мультипроцесорних системах) та інших ресурсів;

- керування пам'яттю: відображення практично необмеженої віртуальної пам'яті процесів у фізичну оперативну пам'ять обмеженого розміру, розподілене використання тих самих областей оперативної пам'яті декількома процесами;

- керування файлами: реалізація абстракції файлової системи, підтримка ієрархії каталогів і файлів; сучасні варіанти ОС UNIX одночасно підтримують кілька типів файлових систем;

- обмін даними між процесами, що виконуються одним комп'ютером (IPC – Inter-Process Communications), між процесами, що виконуються в різних вузлах локальної чи глобальної мережі передачі даних, а також між процесами та драйверами зовнішніх пристроїв;

- забезпечення процесам користувача доступу до можливостей ядра на основі механізму системних викликів, оформлених у вигляді бібліотеки функцій.

Одна з основних переваг ОС UNIX – її висока мобільність, тобто операційна система досить просто переноситься на різні апаратні платформи. Усі частини системи, окрім ядра, є цілком апаратно-незалежними. Ці компоненти написані мовою С, і для їхнього перенесення на нову апаратну платформу потрібна тільки перекомпіляція вихідних текстів у коди цільового комп'ютера.

Ядро системи цілком приховує апаратні особливості комп'ютера і складається з апаратно-залежних і апаратно-незалежних компонентів. Основна апаратно-незалежна частина ядра написана мовою С і для переносу на нову платформу потребує тільки перекомпіляції. Лише невелика апаратно-залежна частина ядра написана мовою С із вставками мовою асемблера цільового процесора.

Апаратно-залежна частина ядра ОС UNIX містить такі компоненти:

- розкручування й ініціалізація системи на низькому рівні;

- первинна обробка внутрішніх і зовнішніх переривань;

- керування пам'яттю (у тій частині, що відноситься до особливостей апаратної підтримки віртуальної пам'яті);

- переключення контексту процесів між режимами користувача і ядра;

- пов'язані з особливостями цільової платформи частини драйверів пристроїв.

1.7.3. Основні підсистеми ядра ОС UNIX

Ядро складається з трьох основних підсистем: файлової підсистеми, підсистеми керування процесами та пам'яттю, підсистеми введення/виведення.

Функції основних підсистем ядра ОС UNIX наведено в табл. 1.2.

Таблиця 1.2

Функції основних підсистем ядра ОС UNIX

Підсистема ядра	*Функції підсистеми*
Файлова підсистема	- операції розміщення і видалення файлів; - запис/читання даних файла; - визначення прав доступу користувачів до файлів; - доступ користувачів до периферійних пристроїв
Підсистема керування процесами	- створення і видалення процесів; - розподіл системних ресурсів між процесами; - синхронізація процесів; - міжпроцесорна взаємодія
Підсистема введення-виведення	- доступ до периферійних пристроїв; - буферизація даних; - взаємодія з драйверами пристроїв, які є спеціальними модулями ядра, що безпосередньо обслуговують зовнішні пристрої

1.7.4. Принципи взаємодії з ядром

В ОС UNIX механізм, що дозволяє програмам користувача звертатися до послуг ядра, оснований на системних викликах. При виконанні системних викликів виникає особливого роду внутрішнє переривання процесора, що переводить його в режим ядра (у більшості сучасних ОС такі види переривань називаються trap - пастка). При обробці таких переривань ядро розпізнає, чи насправді переривання є запитом до ядра з боку програми користувача на виконання визначених дій, вибирає параметри звернення й обробляє його, після чого відновлює нормальне виконання програми.

Конкретні механізми генерування внутрішніх переривань з ініціативи програми користувача залежать від апаратних особливостей комп'ютера. В ОС UNIX є додатковий рівень, що приховує особливості конкретного механізму генерування внутрішніх переривань. Цей механізм забезпечується бібліотекою системних викликів.

Для користувача бібліотека системних викликів являє собою звичайну бібліотеку заздалегідь реалізованих функцій системи програмування мови С. При програмуванні мовою С використання будь-якої функції з бібліотеки системних викликів нічим не відрізняється від використання будь-якої власної чи бібліотечної С-функції. Однак усередині будь-якої функції конкретної бібліотеки системних викликів міс-

титься програмний код, що є специфічним для цієї апаратної платформи.

1.7.5. Принципи обробки переривань

Суть механізму обробки переривань полягає в тому, що кожному можливому перериванню процесора (внутрішньому чи зовнішньому) відповідає фіксована адреса фізичної оперативної пам'яті, де розташована програма його обробки. Ця адреса називається *вектором переривання*. При обробці процесором переривання відбувається апаратна передача керування на програму його обробки.

У векторі переривання, що відповідає зовнішньому перериванню, тобто перериванню від зовнішнього пристрою, містяться команди, які встановлюють його рівень пріоритету і здійснюють перехід на програму повної обробки переривання у відповідному драйвері пристрою. Для внутрішнього переривання (наприклад, переривання з ініціативи програми користувача за відсутності в основній пам'яті потрібної сторінки, при виникненні виняткової ситуації в програмі користувача і т.д.) чи переривання від таймера у векторі переривання міститься перехід на відповідну програму ядра ОС UNIX.

1.8. Висновки

Розгляд цієї теми включає:

1) короткий нарис історії створення і розвитку ОС UNIX, а також аналіз причин її популярності;

2) базові поняття ОС UNIX і узагальнену архітектуру операційної системи, яка базується на дворівневій моделі;

3) структуру ядра ОС UNIX і його основні підсистеми;

4) принципи взаємодії програм користувача з ядром і спосіб обробки переривань.

2. СЕРЕДОВИЩЕ КОРИСТУВАЧА ОС UNIX

2.1. Традиційний інтерфейс користувача ОС UNIX

ОС UNIX є типовою інтерактивною операційною системою.

У перших версіях ОС UNIX єдиним апаратним засобом інтерактивної взаємодії з обчислювальною системою були алфавітно-цифрові термінали. Тому історично базовим засобом взаємодії сис-

теми з користувачем є рядковий інтерфейс, що функціонує згідно з таким алгоритмом:

- користувач вводить зі свого термінала деякий рядок символів;
- система контролює синтаксис цього рядка;
- якщо рядок розпізнається системою, то система виконує відповідні дії і видає на екран користувача отримані результати; якщо рядок символів система не розпізнає, на екран виводиться повідомлення про помилку.

Широке поширення персональних комп'ютерів із графічними інтерфейсами стимулювало перехід ОС UNIX на використання графічних інтерфейсів взаємодії з користувачем. В усіх сучасних версіях ОС UNIX підтримується можливість використання графічних терміналів у багатовіконному режимі з застосуванням відповідного графічного інтерфейсу в кожнім вікні.

Але в будь-якій модифікації ОС UNIX хоча б одне вікно графічного термінала використовується як аналог традиційного алфавітно-цифрового термінала для взаємодії з системою в традиційному режимі. Це – *базове середовище користувача*. Багато професійних програмістів воліють використовувати традиційні інтерфейси, що забезпечують великі можливості при малих накладних витратах. Тому на сьогодні без знання основ традиційного інтерфейсу з ОС UNIX обійтися усе ще неможливо.

2.2. Командні мови і командні інтерпретатори

Традиційний інтерфейс користувача ОС UNIX базується на використанні командних мов при роботі в інтерактивному режимі; кожен рядок, що вводиться з термінала і відправляється системі, інтерпретується системою як команда користувача системі. Командні мови ОС UNIX є добре визначеними і містять багато засобів, що наближають їх до мов програмування.

Командні мови операційної системи належать до сім'ї інтерпретованих мов, на відміну від компілятивних, якими є більшість мов високого рівня. Мова називається *компилятивною*, якщо будь-яка закінчена конструкція мови є настільки замкнутою, що забезпечує можливість ізольованої обробки без потреби залучення додаткових мовних конструкцій. В іншому випадку розуміння мовної конструкції не гарантується. Основною перевагою *інтерпретованих* мов є те, що у випадку їх використання програма створюється в покроковому режимі, тобто людина приймає рішення про свій наступний крок залежно від

реакції системи на попередній крок. Командні мови не використовуються для програмування в звичайному розумінні цього слова, хоча на розвинутій командній мові можна написати будь-яку програму. Командна мова застосовується в основному для безпосередньої взаємодії із системою з залученням можливостей складання командних файлів (скриптів чи сценаріїв у термінології ОС UNIX) для економії часу при реалізації повторюваних рутинних процедур.

Програми, призначені для обробки конструкцій командних мов і виконання команд користувача, називаються *командними інтерпретаторами*. Командна мова, як правило, нерозривно зв'язана з відповідним інтерпретатором. ОС UNIX зазвичай поставляється з чотирма командними інтерпретаторами сім'ї shell: Bourne-shell, C-shell, Korn-shell і Bourne-Again shell, який в останній час має значне поширення.

У командних мовах існує чотири види команд:

- власні (вбудовані) команди shell (наприклад, **`cd, echo, exec`** і т.д.) являють собою частину програмного коду командного інтерпретатора; ці команди визначені в командній мові, їх неможливо змінити без переробки інтерпретатора;

- зовнішні бібліотечні команди (наприклад, **`find, grep, cc`** і т.д.) складають частину системного програмного забезпечення, вони є іменами файлів, що містять виконувані програми (утиліти), набір яких поставляється з системою;

- користувальницькі команди – це будь-яка виконувана програма, організована відповідно до вимог системи; ОС UNIX може необмежено розширювати спектр зовнішніх команд своєї командної мови (наприклад, можна написати власний командний інтерпретатор);

- командні файли пишуться на самій мові shell і являють собою окремі файли, що містять послідовність рядків у синтаксисі командної мови.

2.3. Базові можливості сім'ї командних інтерпретаторів

Командний інтерпретатор відіграє важливу роль в ОС UNIX, забезпечуючи функціонування базового середовища користувача та реалізуючи такі функції:

- ініціалізацію операційної системи;

- забезпечення входу користувача в систему і виконання рутинних ініціалізаційних дій;

- надання зручного засобу програмування.

2.3.1. Ініціалізація операційної системи

Ініціалізація ОС UNIX забезпечується шляхом виконання відповідних ініціалізаційних скриптів. Командний інтерпретатор дозволяє модифікувати ці скрипти, додаючи новий сервіс у процес ініціалізації.

2.3.2. Забезпечення сеансу роботи користувача в системі

Сеанс роботи користувача в ОС UNIX реалізується згідно з таким алгоритмом:

- активізація процесу термінального доступу (процесу **`getty`**) після включення користувачем термінала;
- запит ім'я і пароля користувача і верифікація користувача (програма **`login`**);
- запуск програми, зазначеної в рядку ідентифікації користувача (у файлі **`/etc/passwd`**); звичайно цією програмою є якийсь командний інтерпретатор сім'ї shell;
- командний інтерпретатор забезпечує інтерфейс командного рядка, зчитуючи введений користувачем рядок, виконуючи його синтаксичний аналіз і підстановку шаблонів, а також задані в командному рядку дії; користувач має можливість запустити з командного рядка як програму, так і внутрішню функцію інтерпретатора;
- вихід користувача з системи (команда **`exit`**) шляхом завершення сеансу роботи з інтерпретатором.

2.3.3. Надання користувачу засобу програмування

Будь-яка з мов сім'ї shell являє собою розвинуту мову програмування, основану на використанні різноманітних утиліт, багато з яких є не менш складними програмами, ніж сам командний інтерпретатор. Програми, написані за допомогою мови shell, є командними файлами (скриптами). При виконанні скрипта командний інтерпретатор shell послідовно обробляє команди, що містяться в скрипті, так, ніби вони вводилися користувачем з термінала. Такий спосіб виконання наочний, простий, але не оперативний і тому непридатний для розробки програм, для яких основною вимогою є висока продуктивність.

2.4. Коротка характеристика командної мови Bourne-Again shell

Bourne-Again shell (bash) є досить розповсюдженою командною мовою (і одночасно командним інтерпретатором) ОС UNIX. У табл. 2.1 наведено основні визначення цієї мови.

Таблиця 2.1

Основні визначення командної мови Bourne-Again shell

Поняття мови	*Визначення поняття*
Пробіл	Символ пробілу, або символ горизонтальної табуляції
Ім'я	Послідовність букв, цифр чи символів підкреслення, що починається з букви чи підкреслення
Параметр	Ім'я, цифра чи один із символів "*", "@", "#", "?", "-", "$", "!"
Коментарі	Усе, що знаходиться в рядку (у скрипті) лівіше символу "#", сприймається інтерпретатором як коментар
Проста команда	Послідовність слів, розділених пробілами. Перше слово простої команди – це її ім'я (нульовий аргумент), інші слова – аргументи команди
Мета-символи	Аргументи команди (які зазвичай задають імена файлів) можуть містити спеціальні символи (метасимволи) ("*", "?"), а також взяті в квадратні дужки списки чи діапазони зазначених символів. У цьому випадку задане текстове представлення параметра називається шаблоном. Указівка символу "*" означає, що замість зазначеного шаблона може використовуватися будь-яке ім'я, у якому цей символ замінено на довільний текстовий рядок. Завдання в шаблоні символу "?" означає, що у відповідній позиції може використовуватися будь-який припустимий символ. Нарешті, при використанні в шаблоні квадратних дужок для генерації імені використовуються всі зазначені в квадратних дужках символи. Команда застосовується в циклі для всіх генерованих імен
Значення простої команди	Код її завершення, якщо команда закінчується нормально, або код помилки (128), якщо завершення команди не нормальне (усі значення видаються в текстовому вигляді)
Команда	Проста команда або одна з керуючих конструкцій (спеціальних убудованих у мову конструкцій, призначених для організації складних bash-програм)

Закінчення табл. 2.1

Поняття мови	*Визначення поняття*
Командний рядок	Текстовий рядок мовою bash
shell-процедура	Файл із програмою, написаною мовою shell
Конвеєр	Послідовність команд, розділених символом "|". При виконанні конвеєра стандартне виведення кожної команди, окрім останньої, направляється на стандартний вхід наступної команди. Інтерпретатор shell очікує завершення останньої команди конвеєра. Код завершення останньої команди вважається кодом завершення усього конвеєра
Список	Послідовність декількох конвеєрів, з'єднаних символами ";", "&", "&&", "||", яка, можливо, закінчується символами ";" чи "&". Роздільник між конвеєрами ";" означає, що потрібно послідовне виконання конвеєрів; "&" означає, що конвеєри можуть виконуватися паралельно. Використання як роздільника символів "&&" ("||") означає, що наступний конвеєр буде виконуватися тільки в тому випадку, якщо попередній завершився нормально (з помилкою). При організації списку символи ";" і "&" мають однакові пріоритети, менші, ніж роздільникі "&&" і "||"

У будь-якому місці програми може бути оголошена (і встановлена) змінна за допомогою конструкції **`ім'я=значення`** (усі значення змінних – текстові). Використання конструкцій **`${ім'я}`** приводить до підстановки поточного значення змінної у відповідне слово.

Виклик будь-якої команди можна обрамити одиночними лапками (`), тоді у відповідний рядок буде підставлений результат стандартного виведення цієї команди.

Командна мова Bourne-Again shell для роботи з текстовими значеннями має такі керуючі конструкції :

- **`for`** і **`while`** – для організації циклів;
- **`if`** – для організації розгалужень;
- case – для організації перемикачів.

2.5. Програмування командною мовою Bourne-Again shell (bash)

Командна мова bash – це мова програмування дуже високого рівня. На цій мові користувач здійснює керування комп'ютером. Ознакою командного рядка (промптером) є зазвичай символ долара (**$**), хоча користувачу надається можливість змінювати промтер.

В ОС UNIX є близько 200 базових команд — інструментальних засобів, що дозволяють користувачу вирішувати багато проблем, не звертаючись до програмування на мовах високого рівня чи до використання спеціальних пакетів.

2.5.1. Найпростіші засоби bash

Найпростішими засобами мови bash є вбудовані команди, бібліотечні команди, команди користувача і скрипти.

Алгоритм обробки команди показано на рис. 2.1.

Рис. 2.1. Алгоритм обробки команди інтерпретатором bash

Команди в bash мають таку *структуру*:

<ім'я команди> <-опції> <аргумент(и)>

Як правило (є й винятки), перше слово (тобто послідовність символів до пробілу, чи табуляції кінця рядка) bash сприймає, як команду.

Наприклад, команда виведення вмісту каталогу **ls**:

```
ls -ls /usr/bin
```

де **ls** - ім'я команди; **-ls** - опції команди (дефіс – ознака опції, **l** – опція, що забезпечує виведення інформації в повному форматі, **s** – опція, що забезпечує виведення з вказівкою розміру файлів у блоках);

/usr/bin – ім'я каталогу, вміст якого видається на екран у повному форматі з додаванням інформації про розмір кожного файла в блоках.

Команда **cat f1** видає на екран вміст файла **f1**, а команда **cat cat** – файл з ім'ям **cat** (друге слово), що знаходиться в активному каталозі.

Мова bash має *засоби групування*, наведені в табл. 2.2.

Таблиця 2.2

Засоби групування командної мови bash

Засіб	*Призначення засобу*
<переведення рядка>	Визначає послідовне виконання команд
&	Задає асинхронне (фонове) виконання попередньої команди. При виконанні команди в асинхронному режимі на екран виводиться номер процесу, що відповідає виконуваній команді, і система, запустивши цей фоновий процес, знову виходить на діалог з користувачем
&&	Забезпечує виконання наступної команди за умови нормального завершення попередньої, інакше наступна команда ігнорується
||	Забезпечує виконання наступної команди при помилковому завершенні попередньої, інакше наступна команда ігнорується
Фігурні "{}" і круглі "()" дужки	Забезпечують групування команд

Наприклад, група команд для запуску команди **find** для пошуку у фоновому режимі файла з ім'ям **conf**, починаючи від кореня, а потім виконання команди **pwd** у звичайному режимі має такий вигляд:

```
$ find / -name conf -print & # уведення команди find.
```

У цьому випадку на екран буде виведений номер фонового процесу, наприклад 288):

```
$ pwd                          #введення команди pwd
/mnt/lab/asu                   ( результат роботи pwd)
$                              (bash-промптер)
/usr/include/sys/conf          (результат роботи find).
```

Для групування команд також можуть використовуватися фігурні "{}" і круглі "()" дужки. Наприклад:

- команда **k1 && k2; k3**, де **k1**, **k2** і **k3** - будь-які команди, виконується так: **k2** буде виконана тільки при успішному завершенні **k1**; після кожного з результатів обробки **k2** (тобто якщо **k2** буде виконана або пропущена) буде виконана **k3**;

- команда **k1 && {k2; k3}** виконується так: обидві команди (**k2** і **k3**) будуть виконані тільки при успішному завершенні **k1**;

- команда **{k1; k2} &** виконується так: у фоновому режимі буде виконуватися послідовність команд **k1** і **k2**.

Команди ОС UNIX, що можуть працювати зі стандартним введенням і виведенням, називаються *фільтрами*.

Зовнішні команди bash, як і будь-які інші команди ОС UNIX, читають вхідні дані зі стандартного введення і виводять свої результати та повідомлення про помилки на стандартне виведення, тобто є фільтрами. Стандартне введення в ОС UNIX здійснюється з клавіатури термінала, а стандартне виведення спрямоване на екран термінала. Приклад переадресування стандартних потоків введення/виведення і організації конвеєра подано на рис. 2.2.

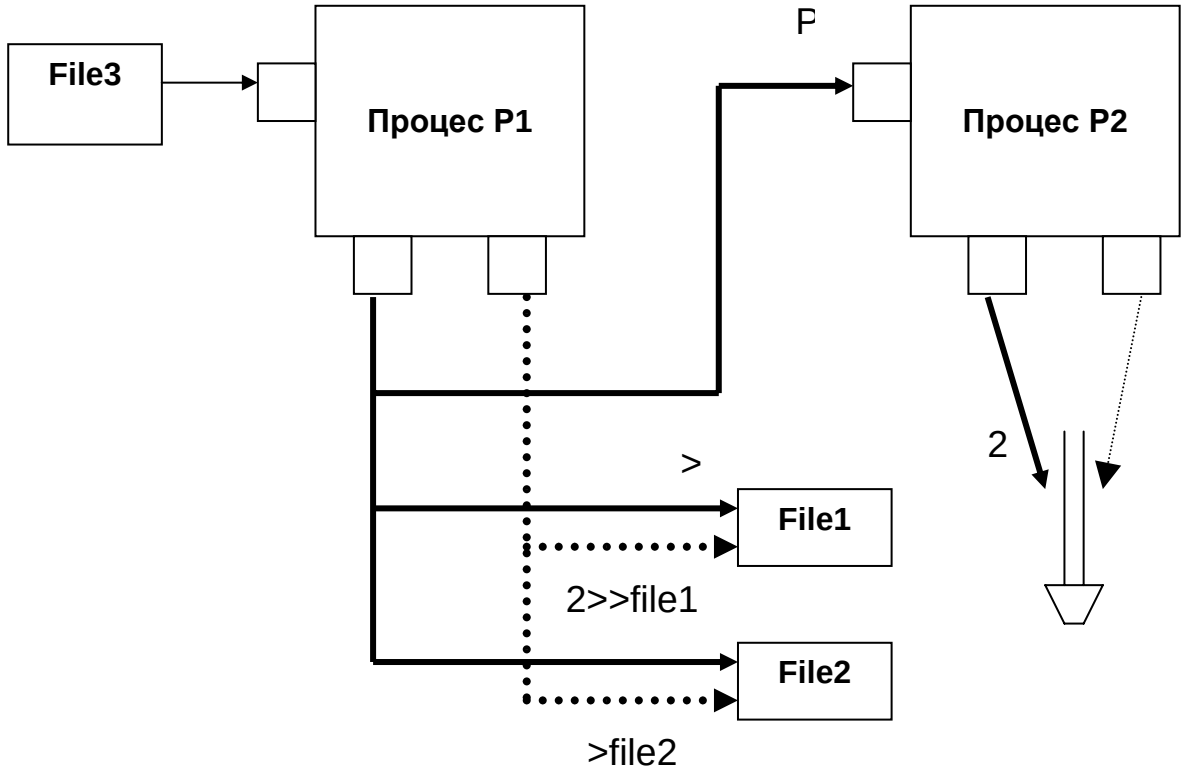


Рис. 2.2. Переадресування стандартних потоків введення/виведення і організація конвеєра в ОС UNIX

Користувач має зручні *засоби переадресування* введення і виведення на інші файли (пристрої).

Символи ">" і ">>" позначають *переадресування виведення*.

Наприклад:

- команда **ls >f1** формує список файлів поточного каталогу та розміщує його у файлі **f1** (замість видачі на екран). Якщо файл **f1** до цього існував, то він буде затертий новим файлом;

- команда **pwd >>f1** формує повне ім'я поточного каталогу і розміщує його в кінці файла **f1**, тобто символи **>>** означають додавання у файл, якщо він не порожній.

Символи "<" і "<<" позначають *переадресування введення*.

Наприклад, команда **wc -l <f1** підраховує і видає на екран число рядків у файлі **f1**.

Конструкція **com1 par1, par2, ..., parn > file_name** в командному рядку означає, що для стандартного виведення команди **com1** буде використовуватися файл з ім'ям **file_name**.

Конструкція **com1 par1, par2, ..., parn < file_name** означає, що команда **com1** буде використовувати файл з ім'ям **file_name** як джерело свого стандартного введення.

Можна сполучити переадресування: команди **wc -l <f3 >f4** і **wc -l >f4 <f3** виконуються однаково: підраховують число рядків файла **f3** і результат розміщують у файлі **f4**.

Засіб, що поєднує стандартний вихід однієї команди зі стандартним входом іншої, називається *конвеєром* і позначається вертикальною рискою "|".

Наприклад, послідовність команд **ls|wc –l** список файлів поточного каталогу спрямовує на вхід команди **wc**, яка на екран виводить число рядків каталогу.

Конвеєром можна поєднувати і більше двох команд, якщо всі вони, крім першої та останньої, – фільтри:

Наприклад, конвеєр команд

cat f1 | grep -h result | sort | cat -b > f2 з файла **f1** (команда **cat**) вибирає всі рядки, що містять слово **result** (команда **grep**), сортує (команда **sort**) отримані рядки, нумерує їх (команда **cat -b**) і виводить результат у файл **f2**.

Оскільки пристрої в ОС UNIX подані спеціальними файлами, їх можна використовувати при переадресуванні. Спеціальні файли знаходяться в каталозі **/dev**: **lp** – друк; **console** – консоль; **ttyi** – і-й термінал; **null** – фіктивний файл чи пристрій.

Команда **ls > /dev/lp** виводить вміст поточного каталогу на друк, а команда **f1 < /dev/null** обнуляє файл **f1**.

Конвеєр команд **sort f1 | tee /dev/lp | tail –20** сортує файл **f1** і передає його на друк, а 20 останніх рядків видаються на екран.

Перенаправляючи виведення у файл фіктивного пристрою **(/dev/null**), можна завадити його відображенню на екрані. Наприклад, при виконанні команди **cat f1 f2** на екран послідовно видається вміст файлів **f1** і **f2**. Якщо вміст файла **f1** є 1111111, а файл **f2** відсутній, то повідомлення на екрані має такий вигляд:

```
111111
cat: f2: No such file or directory
```

(команда **cat** видала повідомлення на стандартний вихід – екран).

Використання команди **cat f1 f2 2>f-err** дозволяє повідомлення про помилки записати у файл **f-err**.

За допомогою конструкції **cat f1 f2 >>ff 2>ff** можна всю інформацію направити в один файл **ff**.

Можна вказати не тільки на те, який зі стандартних файлів переадресовувати, але й у який стандартний файл здійснити переадресування. Конструкція **cat f1 f2 2>>ff 1>&2** спочатку спрямовує стандартний потік помилок (у режимі додавання) у файл **ff**, а потім потік результату перенаправляє на стандартний вихід, яким до цього моменту є файл **ff**, тобто результат буде аналогічний попередньому. Конструкція **1>&2** указує на злиття потоків з дескрипторами 1 і 2.

Конвеєр – це простий, але винятково могутній засіб мов сім'ї shell, бо дозволяє під час роботи динамічно створювати комбіновані команди. Наприклад, вказівка в одному командному рядку послідовності зв'язаних конвеєром команд **ls -l | sort –r** забезпечує видачу на екран вмісту поточного каталогу (команда **ls -l**), що буде відсортований за іменами файлів у зворотному порядку (команда **sort –r**). Якби не було можливості комбінування команд, то для досягнення такої можливості необхідно було б впровадження в програму **ls** можливостей сортування.

ОС UNIX надає засоби для *генерації імен файлів*, використовуючи метасимволи, подані в табл. 2.1. Приклад генерації імен файлів із використанням метасимволів подано у табл. 2.3.

Таблиця 2.3

Приклади генерації імен файлів

Команда	*Опис команди*
cat f*	Видає усі файли каталогу, що починаються з символу **f**
cat *f*	Видає усі файли, що містять **f**
cat program.?	Видає файли даного каталогу з односимвольними розширеннями, наприклад **program.c** і **program.o**, але не видасть файл **program.com**
cat [a-d]*	Видає файли, що починаються із символів a, b, c, d. Аналогічно працюють і команди **cat [abcd]*** і **cat [bdac]***

2.5.2. Загальний синтаксис скрипта

Скрипт – це звичайний текстовий файл, кожен рядок якого являє собою інструкцію (команду чи функцію), що виконується командним інтерпретатором. В ОС UNIX можуть існувати скрипти для різних командних інтерпретаторів, тому у першому рядку скрипта вказується ім'я інтерпретатора. Перший рядок скрипта для командного інтерпретатора bash має такий вигляд: **#!/bin/bash.**

2.5.3. Змінні й параметри в командній мові bash

Shell-програма при виконанні може використовувати змінні, як визначені раніше, так і самостійно встановлені, які використовуються ідентично. Специфікою командних інтерпретаторів сім'ї shell (зв'язаною з їх орієнтацією на забезпечення інтерфейсу з операційною системою) є те, що всі змінні shell-програми (сеансу виконання командного інтерпретатора) є текстовими, тобто містять рядки символів.

У командному рядку чи в скрипті можна використовувати змінні таких типів: звичайні, системні, вбудовані, глобальні.

Значення ***звичайної змінної*** можна задати трьома способами:

- за допомогою оператора присвоювання;
- шляхом присвоєння значення з потоку виведення команди;
- з клавіатури за допомогою команди read.

При присвоюванні значень змінним *за допомогою оператора присвоєння* (=) як змінна, так і її значення мають бути записані без

пробілів щодо символу “=”: **var=value**, де **var** – ім'я змінної, **value** – її значення.

Наприклад: **var1=13**, де **13** – це не число, а рядок із двох цифр; **var2="ос UNIX"** – тут подвійні лапки (" ") необхідні, оскільки в рядку є пробіл.

Значення змінної можна одержати, використовуючи символ $. Наприклад, **echo $var** – виведення значення змінної **var** на екран; **var1=$var** - присвоєння змінній **var1** значення змінної **var**.

Мова bash дозволяє присвоїти *змінній значення з потоку виведення команди*. Наприклад:

- при виконанні інструкції **DAT=`date`** спочатку виконується команда **date** (зворотні лапки свідчать про те, що спочатку має бути виконана укладена в них команда), а результат її виконання, замість видачі на стандартне виведення, приписується як значення змінної, у даному випадку **DAT**;

- при виконанні інструкції **pwd** на екран виводиться ім'я поточного каталогу; при виконанні інструкції **cdir=`pwd`** змінній **cdir** присвоюється ім'я поточного каталогу.

Існують і більш складні умовні конструкції присвоєння значень змінним, нижче наведені деякі з них:

- **$(var:=string)** – присвоєння значення **var**, якщо цю змінну визначено; якщо змінну **var** не визначено, їй присвоюється значення **string**;

- **$(var:-string)** – присвоєння значення **var**, якщо цю змінну визначено, а якщо змінну **var** не визначено, то значення **string**; значення **var** при цьому не змінюється;

- **$(var:+string)** – присвоєння значення **string**, якщо змінну **var** визначено, в іншому випадку – нічого не присвоюється;

- **$(var:?string)#** – якщо змінну **var** не визначено, виводиться рядок **string** й інтерпретатор припиняє роботу, а якщо рядок **string** порожній, то виводиться повідомлення: **parameter not set** (параметр не встановлений).

Можна присвоїти значення змінній за допомогою команди **read**, яка забезпечує введення значення змінної з клавіатури в діалоговому режимі. Зазвичай команді **read** у командному файлі передує команда **echo**, що дозволяє попередньо видати на екран запит на введення. Наприклад:

```
echo -n "Уведіть тризначне число:"
read x
```

При виконанні цього фрагмента командного файла після виведення на екран повідомлення “Уведіть тризначне число:” інтерпретатор буде чекати введення значення з клавіатури. Якщо ввести, наприклад **753,** то це і стане значенням змінної **x**.

Одна команда **read** може прочитати (присвоїти) значення відразу декільком змінним. Якщо змінних у **read** більше, ніж їх введено (через пробіли), тим, що залишилися, присвоюється порожній рядок. Якщо переданих значень більше, ніж змінних у команді **read**, то зайві ігноруються.

Щоб ім'я змінної не зливалося з символами, які розташовані за її ім'ям, використовують фігурні дужки. Наприклад, якщо **a=/mnt/lab/asu/,** то інструкції **cat /mnt/lab/asu/prim** і **cat ${a}prim** рівноцінні (тобто **cat** видасть на екран вміст того самого файла).

Якщо припустити, що в системі є змінна **prim** і **prim=dir** , то команда **echo ${a}$prim** видасть на екран **/mnt/lab/asu/dir.**

ОС UNIX має *засоби екранування:* подвійні лапки (" "), одинарні лапки (' ') і бек-слеш (\).

З наведених нижче прикладів очевидна їх дія. (В одному рядку можна записувати декілька операторів присвоєння):

```
x=22 y=33 z=$x
A="$x" B='$x' C=\$x
D="$x+$y+$z" E='$x+$y+$z' F=$x\+\$y\+\$z
```

Тоді команди

```
echo A=$A B=$B C=$C
echo D=$D E=$E F=$F
eval echo evaluated A=$A
eval echo evaluated B=$B
eval echo evaluated C=$C
```

видадуть на екран таку інформацію:

```
A=22 B=$x C=$x
D=22+33+22 E=$x+$y+$z F=22+33+22
evaluated A=22
evaluated B=22
```

У трьох останніх випадках використана команда **eval** (від evaluate – позначати), що у підставленій у ній як аргумент команді виконує означення змінних (якщо такі є). У результаті значення A залишається попереднім, а змінні **B** і **C** мають значення **$x**. Завдяки означенню, що було виконано командою **eval, B** і **C** набувають значення **22**.

Ще приклад: **w=\$v v=\$u u=5.** У результаті виконання команд

```
echo $w
eval echo $w
eval eval echo $w
```

на екран буде виведено

```
$v
$u
5.
```

Бек-слеш (\) не тільки екранує наступний за ним символ, що дозволяє використовувати спеціальні символи просто як символи, які позначають самих себе (він може екранувати і сам себе – \\); у командному файлі бек-слеш дозволяє поєднувати рядки в один (екранувати кінець рядка). Наприклад, командний рядок

```
cat f1 | grep -h result | sort | cat -b > f2
```

може бути записаний в командному файлі так:

```
cat f1 | grep -h  \
result | sort | cat -b > f2
```

Слід зазначити, що ефект продовження командного рядка забезпечує і символ конвеєра, тобто рядок можна записати і так:

```
cat f1          |
grep -h result  |
sort            |
cat -b > f2
```

Незважаючи на те, що командним інтерпретатором змінні в загальному випадку сприймаються як рядки (тобто 35 – це не число, а рядок із двох символів 3 і 5), у деяких випадках змінні можуть інтерпретуватися як цілі числа.

Різноманітні можливості має команда **expr**, ілюстрація її роботи надана нижче: виконання командного файла:

```
x=7 y=2
a=`expr $x + $y`   ; echo a=$a
a=`expr $a + 1`    ; echo a=$a
b=`expr $y - $x`   ; echo b=$b
c=`expr $x '*' $y` ; echo c=$c
d=`expr $x / $y'   ; echo d=$d
e=`expr $x % $y'   ; echo e=$e
```

видасть на екран

```
a=9
a=10
b=-5
c=14
d=3
e=1
```

Примітка. Операція множення (*) обов'язково має бути заекранована, оскільки в bash цей значок сприймається як метасимвол маски.

З командою **expr** можливі не тільки цілочисельні арифметичні операції, але й рядкові:

```
A=`expr 'cocktail' : 'cock'`  ; echo $A
B=`expr 'cocktail' : 'tail'`  ; echo $B
C=`expr 'cocktail' : 'cook'`  ; echo $C
D=`expr 'cock' : 'cocktail'`  ; echo $D
```

Команда **echo** на екран виводить числа, які показують кількість символів у ланцюжках (від початку), що збігаються. Другий з рядків не може бути довшим за перший:

```
4
0
0
0
```

Для нормальної роботи ОС UNIX має бути визначений ряд ***системних змінних***. Нижче наведено деякі з них: **HOME** – ім'я каталогу верхнього рівня, встановленого для користувача; **PATH** – шлях пошуку файла (тропа), якщо в його імені не вказується шлях; **TERM** – ім'я термінала користувача; **LOGNAME** – ім'я користувача; **PSI** – символ промтера; **Shell** – командний інтерпретатор, що використовується; **MAIL** – місце розташування поштової скриньки користувача.

Інформацію про системні змінні, створені при вході в систему і передані усім процесам користувача на правах успадкування, можна одержати за допомогою команди **set**.

В ОС UNIX існує поняття процесу. Процес виникає тоді, коли запускається на виконання будь-яка команда чи прикладна програма.

У кожного процесу є своє середовище – безліч доступних йому змінних. Змінні локальні в рамках процесу, в якому вони оголошені, тобто де їм присвоєні значення. Щоб змінні, визначені в рамках одного процесу, були доступними для інших процесів, необхідно оголосити їх ***глобальними змінними*** за допомогою вбудованої команди **export**. Так, після інструкції **extern var** змінна **var** буде доступна й для інших процесів у рамках сеансу користувача.

Крім змінних, що встановлюються явно, у мові shell існують змінні, значення яких встановлюються самим інтерпретатором; ці змінні називаються ***вбудованими***, використовуються в скриптах і не існують поза контекстом одержання їх значення. Нижче наведено ряд вбудованих змінних мови bash:

$1 - $9 – позиційні параметри скрипта;

$# – число позиційних параметрів скрипта;

$? – код повернення процесу, що виконувався останнім;

$$ – ідентифікатор поточного процесу;

$! – ідентифікатор останнього процесу, запущеного у фоновому режимі;

**$*** – усі параметри, передані скрипту; параметри передаються як єдине слово, взяте в лапки: “$*”=”$1 $2 …”;

**$@** – усі параметри, передані скрипту; параметри передаються як окремі слова, взяті в лапки: “$@”=”$1” “$2” …”...

У командному інтерпретаторі bash використовуються позиційні параметри (тобто притаманна послідовність їхнього проходження). У командному файлі відповідні параметрам змінні (аналогічно bash-змінним) починаються із символу **$**, а далі – одна з цифр від 0 до 9.

Наприклад, якщо скрипт **examp_1** викликається з параметрами **cock** і **tail**, то ці параметри потрапляють у нове середовище під стандартними іменами **1** і **2**. У вбудованій змінній з ім'ям **0** зберігається ім'я самого скрипта. При звертанні до параметрів перед цифрою ставиться символ долара **$** (як і при звертанні до значень змінних): 0 – відповідає імені даного скрипта; 1 – перший за порядком параметр; 2 – другий параметр і т.д.

Нехай скрипт з ім'ям **examp_1** має вигляд

```
echo Це розрахунок $0:
sort $2 >> $1
cat $1,
```

а файли **cock** і **tail** відповідно містять

```
cock:
Це відсортований файл:
```

і

```
tail:
     1
     3
     2
```

Тоді після виклику команди **examp-1 cock tail** на екран буде виведено:

```
Це розрахунок examp_1:
Це відсортований файл:
1
2
3
```

Оскільки кількість змінних, у які можуть передаватися параметри, обмежена цифрою 9, то для передачі більшої кількості параметрів використовується спеціальна команда **shift**, що здійснює зсув параметрів на 1.

Розглянемо її дію на прикладі.

Нехай командний файл **many** викликається з 13-ю параметрами: **many 10 20 30 40 50 60 70 80 90 100 110 120 130.**

У результаті першого застосування команди **shift** другий параметр розрахунку викликається як **$1**, третій параметр викликається як **$2**, ... , десятий параметр, що був спочатку недоступний, викликається як **$9**. При цьому стає недоступним перший параметр.

Команда **set** також установлює значення параметрів. Наприклад, фрагмент скрипта

```
set a b з
echo перший=$1 другий=$2 третій=$3
```

видасть на екран: **перший=a другий=b третій=c.**

Команда **set** дозволяє також здійснювати контроль виконання програми, наприклад:

set -v – на термінал виводяться рядки, що читаються bash;
set +v – скасовує попередній режим;
set -x – на термінал виводяться команди перед виконанням;
set +x – скасовує попередній режим.

2.5.4. Додаткові конструкції мови bash

Мова bash має ряд додаткових конструкцій, використання яких дозволяє перетворити скрипти в розвинуті програми.

Оператор test [] перевіряє виконання деякої умови. З використанням цієї вбудованої команди формуються оператори вибору та циклу.

Два можливих формати команди: **test умова** чи **[умова]**

Примітка. Між дужками й умовою, що міститься в них, а також між значеннями і символом порівняння чи операції обов'язково мають бути пробіли.

В інтерпретаторі bash використовуються такі умови перевірки файлів:

-f file – файл **file** є звичайним файлом;
-d file – файл **file** – каталог;
-c file – файл **file** – спеціальний файл;
-r file – є дозвіл на читання файла **file**;
-w file – є дозвіл на запис у файл **file**;
-s file – файл **file** не порожній.

У командному інтерпретаторі bash використовуються такі умови перевірки рядків:

str1 = str2 – рядки **str1** і **str2** збігаються;
str1 != str2 – рядки **str1** і **str2** не збігаються;
-n str1 – рядок **str1** існує (не порожній);
-z str1 – рядок **str1** не існує (порожній)

і такі умови порівняння цілих чисел:

x -eq y – x дорівнює y;
x -ne y – x не дорівнює y;
x -gt y – x більше y;
x -ge y – x більше чи дорівнює y;
x -lt y – x менше y;
x -le y – x менше чи дорівнює y.

У цьому випадку команда **test** сприймає рядки символів як цілі числа.

Складні умови в командному інтерпретаторі bash реалізуються за допомогою типових логічних операцій:

! – (not) інвертує значення коду завершення;
-o – (or) відповідає логічному "АБО";
-a – (and) відповідає логічному "І".

У загальному випадку ***умовний оператор if*** має таку структуру:

```
if умова
then список
   [elif умова
   then список]
   [else список]
fi
```

Тут **elif** – скорочений варіант від **else if** і може бути використаний поряд з повним, тобто допускається вкладення довільного числа операторів **if** (як і інших операторів). Конструкції

```
[elif умова
  then список]
```

та

```
[else список]
```

не є обов'язковими (в цьому випадку для вказівки на необов'язковість конструкцій використані квадратні дужки).

Найпростіша структура цього оператора має такий вигляд:

```
if умова
   then список команд
fi
```

Якщо умову виконано, то виконується зазначений список команд, інакше він ігнорується.

Нижче наведено приклади застосування оператора **if**:

Приклад 1. Нехай є скрипт **if-1**:

```
if [ $1 -gt $2 ]
   then pwd
   else echo $0 : Hello!
fi
```

Тоді виклик цього скрипта **if-1 12 11** видасть на екран ім'я поточного каталогу (наприклад, **/home/sae/STUDY/SHELL**), а виклик **if-1 12 13** видасть **if-1 : Hello!.**

Приклад 2. У скрипті можна використовувати той факт, що команди спроможні видавати різний код завершення. Наприклад, нехай є скрипт **if-2:**

```
if a=`expr "$1" : "$2"`
   then echo then a=$a code=$?
   else echo else a=$a code=$?
fi
```

Тоді його виклик **if-2 by by** дасть **then a=2 code=0**, а виклик **if-2 by be** дасть **else a=0 code=1**.

Приклад 3 являє собою приклад на вкладеність:

```
# if-3: Оцінка досягнень
echo -n " А яку оцінку одержав на іспиті?: "
read z
if [ $z = 5 ]
   then echo  Молодець !
   elif [ $z = 4 ]
      then echo  Усе одно молодець !
      elif [ $z = 3 ]
         then echo  Міг би краще !
         elif [ $z = 2 ]
           then echo  Ганьба!
           else echo   !
fi
```

Оператор вибору case має таку структуру:

```
case рядок in
   шаблон)  список команд;;
   шаблон)  список команд;;
   ...
esac
```

Тут **case, in** та **esac** – службові слова. **Рядок** (це може бути й один символ) зіставляється із **шаблоном**. Потім виконується **список команд** вибраного рядка. Службове слово **esac** необхідне для завершення структури. Кінець рядків кожної альтернативи ідентифікується символами **;;**. Кожна альтернатива може складатися з декількох команд. Якщо ці команди будуть записані в один рядок, то як роздільник між ними використовується символ **;**. Останній рядок вибору має шаблон *****, що в структурі **case** вказує будь-яке значення. Цей рядок вибирається, якщо не відбувся збіг значення **рядка** з жодним із раніше записаних шаблонів, обмежених дужкою **)**. Значення проглядають в порядку запису.

Приклад 4.

```
echo -n "А яку оцінку одержав на іспиті?: "
read z
case $z in
5) echo Молодець ! ;;
4) echo Усе одно молодець ! ;;
3) echo Міг би і краще ! ;;
2) echo Ганьба ! ;;
*) echo  ! ;;
esac
```

Приклад 5. Використання структури **case** для реалізації найпростішого меню.

```
# Реалізація меню за допомогою команди case
echo "Назвіть файл, а потім (через пробіл)
наберіть цифру, що відповідає необхідній обробці:
   1 - відсортувати
   2 - видати на екран
   3 - визначити кількість рядків  "
read x y  # x - ім'я файла, y – пункт меню
case $y in
   1) sort < $x ;;
   2 cat < $x ;;
   3) wc -l < $x ;;
   *) echo "Такої команди немає!" ;;
esac
```

Приклад 6. Використання структури **case** у скрипті **case-4**, що додає інформацію до файла, зазначеного першим параметром зі стандартного входу (якщо параметр один), або (якщо два параметри) з файла, зазначеного як перший параметр:

```
# Додавання у файл.
# Використання стандартної змінної.
# "$#" - кількість параметрів.
# ">>" - переадресування з додаванням у файл
case $# in
   1) cat >> $1 ;;
   2) cat >> $2 < $1 ;;
   *) echo "Формат: case-4 [відкіля] куди" ;;
esac
```

Параметр **$1** (при **$#=1**) – це ім'я файла, у якого відбувається додавання зі стандартного входу; параметри **$1** і **$2** (при **$#=2**) – це імена файлів, з якого (**$1**) і в який (**$2**) здійснюється додавання.

В усіх інших випадках (*****) видається повідомлення про те, яким має бути правильний формат команди.

Оператор циклу з перерахуванням for має таку структуру:

```
for ім'я [in список значень]
   do
      список команд
   done
```

де **for** – службове слово, що визначає тип циклу; **do** і **done** – службові слова, що виділяють тіло циклу. Фрагмент **in список значень** може бути відсутнім.

Приклад 7. Нехай скрипт **lsort** має такий вигляд:

```
for i in f1 f2 f3
   do
      proc-sort $i
   done
```

Тут ім'я **i** відіграє роль параметра циклу. Це ім'я можна розглядати як bash-змінну, якій послідовно присвоюються значення (**i=f1**, **i=f2, i=f3**), і команда "**procsort**" виконується в циклі.

Часто використовується форма **for i in ***, що означає "для всіх файлів поточного каталогу".

Приклад 8. Нехай скрипт **proc-sort** має такий вигляд:

```
cat $1 | sort | tee /dev/lp > ${1}_sorted,
```

тобто послідовно сортуються зазначені файли, результати сортування виводяться на друк (**/dev/lp**) і направляються у файли **f1_sorted, f2_sorted** і **f3_sorted.** Можна зробити більш універсальний скрипт **lsort**, якщо не фіксувати перелік файлів у команді, а передавати довільну їх кількість параметрами:

```
for i
   do
    proc-sort $i
   done
```

Тут відсутність після **i** службового слова **in** з переліченням імен свідчить про те, що список надходить через параметри команди. Результат попереднього прикладу можна одержати, якщо набрати:

```
lsort f1 f2 f3
```

Приклад 9.

```
# Видача імен усіх підкаталогів каталогу $dir
   for i in $dir/*
      do
         if [ -d  $i ]
            then echo $i
         fi
      done
```

Приклад 10. Спосіб повторення тих самих дій кілька разів. Змінна **i** набуває тут п'яти значень: 1, 2, 3, 4, 5, але усередині циклу ця змінна відсутня, і тому її значення ніякої ролі не відіграє і нічого не змінює. У результаті п'ять разів повторене те саме обчислення вмісту циклу без змін.

```
# Організації п'ятикратного виконання команди
   for i in 1 2 3 4 5
      do
         cat file-22 > /dev/lp
      done
```

Приклад 11. Скрипт **print-n** ілюструє ще одну корисну можливість у використанні циклу **for**. Тут після **for i** . відсутні **in** і перелік імен, тобто переліком імен для **i** стає перелік параметрів, а отже кількість екземплярів, що друкуються, можна змінювати.

```
 # Завдання числа копій через параметри
 for i
       do
         cat file-22 > /dev/lp
      done
```

Зміст не зміниться, якщо перший рядок розрахунку записати як **for i in $*,** оскільки значення **"$*"** є список значень параметрів.

Оператор циклу з істинною умовою while забезпечує виконання розрахунків, коли заздалегідь невідомий точний список значень параметрів або цей список має бути отриманий в результаті обчислень у циклі.

Оператор циклу **while** має таку структуру:

```
while умова
   do
      список команд
   done
```

де **while** – службове слово, що визначає тип циклу з істинною умовою.

Список команд у тілі циклу (між **do** і **done**) повторюється доти, поки зберігається істинність умови (тобто код завершення останньої команди в тілі циклу дорівнює **0**) чи цикл не буде перерваний зсередини спеціальними командами (**break, continue** чи **exit**). При першому вході в цикл умова має виконуватися.

Приклад 12.

```
# Структура "while"
# Друкування 50 екземплярів файла "file-22"
n=0
while [ $n -lt 50 ]    # поки n < 50
   do
      n=`expr $n + 1`
      cat file-22 > /dev/lp
   done
```

Змінній **n** спочатку присвоюється значення **0**, а не порожній рядок, оскільки команда **expr** працює з bash-змінними як з цілими числами, а не як з рядками. Інструкція **n=`expr $n + 1`** при кожнім виконанні збільшує значення **n** на одиницю.

Приклад 13. Скрипт **pr-br** ілюструє переривання нескінченного циклу командою **break**:

```
# Структура "while" с "break"
# Друкування 50 екземплярів файла "file-22"
n=0
while true
   do
      if [ $n -lt 50 ] # якщо n < 50
         then n=`expr $n + 1`
         else break
      fi
   cat file-22 > /dev/lp
done
```

Команда **break [n]** дозволяє виходити з циклу. Якщо **n** відсутня, то це еквівалентно **break 1**. Параметр **n** вказує на кількість

вкладених циклів, з яких треба вийти, наприклад, **break 3** – це вихід із трьох вкладених циклів.

На відміну від команди **break** команда **continue [n]** лише припиняє виконання поточного циклу і повертає на початок циклу. Вона також може мати параметр. Наприклад, **continue 2** означає вихід на початок другого вкладеного циклу.

Команда **exit [n]** дозволяє вийти з процедури з кодом повернення **0** чи **n** (якщо параметр **n** зазначений). Цю команду можна використовувати не тільки в циклах. Навіть у лінійній послідовності команд вона може бути корисною, щоб припинити виконання поточного розрахунку в заданій точці.

Оператор циклу з помилковою умовою until має таку структуру:

```
until умова
   do
      список команд
   done
```

де **until** – службове слово, що визначає тип циклу з помилковою умовою. Список команд у тілі циклу (між **do** і **done**) повторюється доти, поки зберігається хибність умови чи цикл не буде перерваний зсередини спеціальними командами (**break**, **continue** або **exit**). При першому вході в цикл умова не повинна виконуватися. Відмінність від оператора **while** полягає у тому, що умова циклу перевіряється на хибність (на ненульовий код завершення останньої команди тіла циклу) після кожного (у тому числі й першого) виконання команд циклу.

Приклад 14.

```
until false
   do
      read x
      if [ $x = 5 ]
         then echo enough ; break
         else echo some more
      fi
   done
```

Тут програма з нескінченним циклом чекає введення слів (повторюючи на екрані фразу **some more**), поки не буде введено **5**. Після цього видається **enough** і команда **break** припиняє виконання циклу.

Приклад 15. Чекання полудня, що ілюструє можливість використання в умові обчислення:

```
until date | grep 12:00:
   do
      sleep 30
   done
```

Тут кожні 30 секунд виконується командний рядок умови. Команда **date** видає поточну дату і час. Команда **grep** одержує цю інформацію через конвеєр і сполучує заданий шаблон **12:00:** з результатом, виданим командою **date**. При розбіжності **grep** видає код повернення **1,** що відповідає значенню **falsh**, і цикл виконує чекання протягом 30 секунд, після чого повторюється виконання умови. Опівдні відбудеться порівняння, умова стане **true**, на екран буде виведено відповідний рядок і робота циклу закінчиться.

2.6. Основні утиліти bash і електронний довідник man

Довідковий матеріал про команди й утиліти ОС UNIX, а також ключі й опції до них наведено в електронному довіднику man, що дозволяє швидко одержати вичерпну інформацію про команди, функції, формати чи типи даних файла.

Увесь довідковий матеріал електронного довідника поділений на такі розділи: прикладні утиліти; системні виклики; бібліотечні функції; спеціальні файли, драйвери пристроїв і апаратне забезпечення; формати різних конфігураційних і системних файлів; додаткова інформація про типи файлових систем, визначення типів даних; адміністративні утиліти.

При зверненні до цього довідника в командному рядку слід ввести його ім'я **man** з назвою команди чи функції, про яку необхідно одержати інформацію. Команда **man man** видає інформацію про порядок користування довідником.

2.7. Висновки

Розгляд цієї теми включає:

1) традиційний інтерфейс користувача ОС UNIX і базові можливості сім'ї командних інтерпретаторів;

2) можливості командних мов як засобу програмування;

3) коротку характеристику і найпростіші засоби командної мови Bourne-Again shell.

3. ФАЙЛОВА ПІДСИСТЕМА

3.1. Структура файлової підсистеми ОС UNIX

Поняття файла є одним з найбільш важливих для ОС UNIX. Усі файли, з якими можуть маніпулювати користувачі, розташовуються у файловій системі, що являє собою просте ієрархічне дерево. У вершинах дерева знаходяться каталоги (використовують також терміни *довідники, директорії*), що містять списки файлів. Ці файли, у свою чергу, можуть бути або каталогами, або звичайними файлами, або спеціальними файлами, що виконують функції різних пристроїв введення-виведення. Типову структуру файлової системи ОС UNIX показано на рис. 3.1.

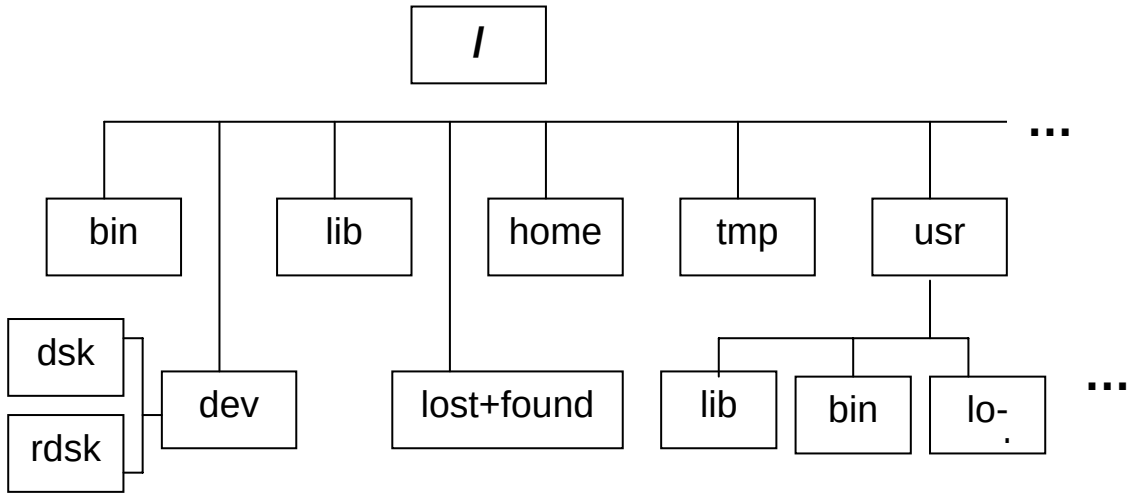


Рис. 3.1. Типова структура файлової системи ОС UNIX

Кожен каталог і файл файлової системи має унікальне *повне ім'я*; в ОС UNIX це ім'я називається *full pathname* – ім'я, що задає повний шлях. Воно вказує повний шлях від кореня файлової системи через ланцюжок каталогів до відповідного каталогу чи файла.

В ОС UNIX використовуються загальноприйняті імена для основних файлів і підкаталогів, що полегшує роботу в операційній системі та забезпечує її перенесення. Каталог, що є коренем файлової системи (кореневий каталог), у будь-якій файловій системі має визначене ім'я / (слеш). У табл. 3.1 подано деякі загальноприйняті імена каталогів.

Таблиця 3.1

Загальноприйняті імена каталогів ОС UNIX

Ім'я каталогу	*Призначення каталогу*
/bin	Каталог для збереження команд і утиліт загального користування
/dev	Каталог для збереження спеціальних файлів, що виконують функції пристроїв (дисплеї, диски, ...)
/lib	Каталог для збереження найважливіших бібліотек
/home	Домашній каталог користувача
/lost+found	Каталог «загублених безіменних» файлів, відновлених ОС
/usr	Каталоги та звичайні файли, що містять інформацію, залучену при розв'язанні задач користувача
/etc	Каталог для збереження команд адміністратора системи
/mnt	Каталог для підключення нових файлових систем
/sys	Засоби для зміни конфігурації системи
/tmp	Каталог для збереження тимчасових файлів

У свою чергу ці каталоги можуть містити каталоги наступного рівня. Наприклад, у каталозі **usr** можуть розташовуватися такі каталоги:

bin – каталог для додаткових команд;
Games – каталог для ігор;
Include – каталог для системних програм;
lib – додаткові бібліотеки.

Повні імена цих каталогів: **/usr/bin**, **/usr/games**, /**usr/include**, **/usr/lib**.

Якщо в каталозі **/usr/include** міститься каталог **sys**, який, у свою чергу, містить каталог **conf**, то повне ім'я каталогу **conf** має такий вигляд: **/usr/include/sys/conf**.

Формальною ознакою повного імені є те, що воно починається зі слеша (/).

Відносне ім'я (relative pathname) визначає ім'я відносно поточного каталогу і не починається із символу /. Наприклад, якщо в даний момент поточним каталогом є каталог **/usr** файлової системи, то відносним ім'ям файла **conf** є **include/sys/conf**.

У кожнім каталозі містяться два спеціальних імені – ім'я "**.**", що іменує сам цей каталог, та ім'я "**..**", що іменує батьківський каталог (який на шляху до кореня знаходиться на сходинку вище даного каталогу).

Як ім'я файла може використовуватися будь-яка послідовність з букв, цифр і підкреслень. Можуть використовуватися й інші символи, однак краще не користуватися спеціальними символами в іменах, оскільки значна частина спеціальних символів мають у командному інтерпретаторі спеціальний вміст.

Довжина імені файла – до 256 символів.

Розширення не є обов'язковими в іменах, хоча ряд команд потребують наявності деяких фіксованих розширень в іменах, наприклад розширення "**.c**" для вихідних файлів для С-компілятора.

В ОС UNIX великі й маленькі букви сприймаються як різні, тому **FILE**, **file** і **File** – це три різних імені.

Окремі частини файлової системи можуть знаходитись на різних фізичних пристроях, наприклад на декількох твердих і гнучких дисках (чи в різних частинах одного диска). Відповідні фрагменти (піддерева файлової системи) монтуються в єдину файлову систему командою **mount** (звичайно це функція адміністратора системи). Тому в ОС UNIX в імені ніяк не відбивається пристрій, на якому файл знаходиться чи створюється.

Командна мова ОС UNIX – bash – дозволяє переглянути вміст каталогу за допомогою команди **ls**. Наприклад, у результаті виконання команди **ls -l /usr**, де **ls** – ім'я команди; **-l** – опція, яка вказує, що видача має бути в довгому форматі; **/usr** – ім'я каталогу, вміст якого виводиться на екран, буде виведене:

```
drwxrwxr-x  2  root  2048  nov 3  12:11  bin
-rwxr--r--  1  root   861  may 11 20:11  boot
drwxrwxr-x  2  root  1024  jan 9  11:55  dev
drwxrwxr-x  1  root  4096  may 11 20:11  dos
drw-r--r--  3  root  4096  nov 17 12:01  include
drwxr-xr-x  7  root  480   nov 17 12:30  lib
```

Запис першого рядку розшифровується так: перший символ у першому рядку означає, що це каталог (**d**-directory); перша тріада (**rwx**) вказує права власника (власнику каталогу дозволяється: **r** – читати, **w** – писати і **x** – виконувати); друга тріада (**rwx**) вказує права членів групи, у яку входить власник (групі також дозволені всі три операції); остання тріада вказує права доступу інших користувачів, яким дозволено тільки читати і виконувати (заборонено писати в цей файл, тобто змінювати вміст каталогу). Далі в рядку символ 2 указує

число зв'язків файла (тобто в системі є ще одне ім'я, зв'язане з цим файлом). Наступне слово в рядку – **root** – вказує ім'я власника, **2048** – число символів у файлі, **nov 3 12:11** – дату і час створення чи останньої модифікації файла (3 листопада в 12-11); **bin** – ім'я файла (у цьому прикладі – каталог команд).

Кожен файл має зв'язані з ним три структури даних:

- ім'я файла, що є атрибутом файлової системи;
- вміст файла – деякий набір даних, що зберігаються на диску;
- метадані файла, що зберігаються в індексних дескрипторах, так званих inode-вузлах (information nodes – інформаційні вузли); кожен inode-вузол має свій унікальний номер (номер inode).

Метадані файла містять інформацію, необхідну операційній системі для виконання операцій над файлом, а саме: тип файла; його атрибути і права доступу до нього; число імен, що має файл у системі; ідентифікатори власника, розмір файла в байтах; час останнього доступу до файла і час останньої модифікації; час останньої модифікації метаданих, а також покажчики на дискові блоки збереження вмісту файла.

3.2. Типи файлів

В ОС UNIX поняття файла є універсальною абстракцією, що дозволяє працювати зі звичайними файлами, що містяться на пристроях зовнішньої пам'яті; із пристроями; з інформацією, динамічно генерованою іншими процесами, і т.д. Для підтримки цих можливостей однаковим способом файлові системи ОС UNIX підтримують такі типи файлів: звичайні файли (regular files); каталоги (directories); спеціальні файли пристроїв (special device files); іменовані канали (named pipes); файли зв'язку (links) та сокети (sockets).

Тип кожного файла відображається при повному лістингу каталогу першим символом першого стовпця рядка, що відповідає файлу.

Звичайні файли являють собою набір блоків на пристрої зовнішньої пам'яті, на якому підтримується файлова система. Такі файли можуть містити як текстову інформацію, так і довільну двійкову інформацію. Файлова система допускає будь-яку структуру цих файлів, розглядаючи їх як послідовність байтів.

Наявності звичайних файлів недостатньо для організації ієрархічних файлових систем. Потрібні *файли-каталоги*, в яких зіставляються імена файлів чи каталогів з їхнім фізичним описом. Каталоги являють собою особливий вид файлів, що зберігаються в зовнішній

пам'яті та мають фіксовану структуру. Фактично, каталог – це таблиця, кожен запис якої складається з двох полів: номера inode-вузла даного файла у файловій системі та імені файла, яке зв'язано з цим номером (цей файл може бути і підкаталогом). У табл. 3.2 наведено приклад вмісту поточного робочого каталогу, отриманого за допомогою команди **`ls - ai`**:

Таблиця 3.2

Приклад вмісту поточного робочого каталогу

Покажчик на метадані файла (номер inode-вузла)	*Ім'я файла (file name)*
33	**.**
122	**..**
54	**first_file**
65	**second_file**
65	**second_again**
77	**dir2**

Цей приклад демонструє, що в будь-якому каталозі містяться два стандартних імені – "**.**" і "**..**". Імені "**.**" відповідає inode-вузол цього самого каталогу, а імені ".." – inode-вузол батьківського каталогу. Файли з іменами **`first_file`** і **`second_file`** – це різні файли з номерами inode-вузлів 54 і 65 відповідно. Файл **`second_again`** описується тим же inode-вузлом, що і файл **`second_file`**, тобто є прикладом так називаного *твердого посилання*: він має інше ім'я, але реально описується тим же inode-вузлом, що і файл **`second_file`**.

Прямий запис інформації у файли-каталоги заборонено, що пояснюється його фіксованою (і закритою від користувачів) структурою. Запис у них здійснюється неявно при створенні та знищенні файлів і підкаталогів, однак читати з файла-каталогу за наявності відповідних прав можна.

Спеціальні файли пристроїв не зберігають даних. Вони забезпечують механізм відображення фізичних зовнішніх пристроїв в іменах файлів файлової системи. Кожному пристрою, який підтримується системою, відповідає, щонайменше, один спеціальний файл. Кожному спеціальному файлу відповідає програмне забезпечення – драйвер відповідного пристрою. При здійсненні операції читання чи запису стосовно спеціального файла виконується прямий виклик відповідного драйвера, програмний код якого відповідає за передачу даних між процесом користувача і відповідним фізичним пристроєм. Імена спеціальних файлів можна використовувати практично всюди, де можна використовувати імена звичайних файлів. Наприклад, команда

cp myfile /kuz/ перепише файл з ім'ям **myfile** у підкаталог **kuz**. А команда **cp myfile /dev/console** видасть вміст файла **myfile** на консоль.

Є два типи спеціальних файлів – блокові й символьні. Блокові спеціальні файли асоціюються з зовнішніми пристроями, обмін з якими здійснюється блоками байтів даних (розміром 512, 1024, 4096 чи 8192 байтів). Типовим прикладом таких пристроїв є магнітні диски. Символьні спеціальні файли асоціюються з зовнішніми пристроями, що не обов'язково потребують обміну блоками даних однакового розміру. Прикладами таких пристроїв є принтери, термінали (у тому числі, консоль), послідовні пристрої, деякі види магнітних стрічок.

При обміні даними з блоковим пристроєм система буферизує дані (використовуючи спеціальний внутрішній системний кеш). Обміни із символьними спеціальними файлами здійснюються прямо, без використання системної буферизації.

Файлова система ОС UNIX забезпечує можливість зв'язку файла з різними іменами на основі створення *спеціальних файлів зв'язку*.

Програмний канал (pipe) – це один із традиційних засобів міжпроцесорної взаємодії в ОС UNIX. Основний принцип роботи програмного каналу полягає в буферизації байтового виведення одного процесу і забезпеченні можливості читання вмісту програмного каналу іншим процесом у режимі FIFO (тобто першим буде прочитаний байт, раніше за всіх записаний). Розрізняються два підвиди програмних каналів – неіменовані й іменовані. *Неіменований програмний канал* створюється процесом-предком, успадковується процесами-нащадками і забезпечує тим самим можливість зв'язку в ієрархії породжених процесів. Інтерфейс неіменованого програмного каналу збігається з інтерфейсом файла. Такі канали не мають імені, їм не відповідає ніякий елемент каталогу у файловій системі.

Іменований програмний канал має вигляд звичайного файла (за ним закріплюється рядок каталогу і власний inode–вузол), але він не містить ніяких даних доти, поки деякий процес не виконає в нього запис. Після того як інший процес прочитає записані в канал байти, цей файл знову стає порожнім. На відміну від неіменованих іменовані програмні канали можуть використовуватися для зв'язку будь-яких процесів (тобто не обов'язково процесів, що входять в одну ієрархію споріднення). Інтерфейс іменованого програмного каналу практично цілком збігається з інтерфейсом звичайного файла, хоча його поведінка є іншою.

Сокети призначені для організації взаємодії між процесами.

3.3. Зв'язування файлів з іменами

З одним inode-вузлом може бути зв'язано кілька імен файлів, тобто архітектура ОС UNIX допускає існування декількох імен для одного файла. Ім'я файла в ОС UNIX є покажчиком на його метадані, але метадані не містять покажчика на ім'я файла. Ім'я жорстко зв'язано з метаданими і, отже, зі вмістом файла, а сам файл існує незалежно від імені, присвоєного йому у файловій системі.

В ОС UNIX підтримуються два типи зв'язку.

Жорсткий зв'язок (рис. 3.2) є звичайним файлом і не належить до будь-якого особливого типу.

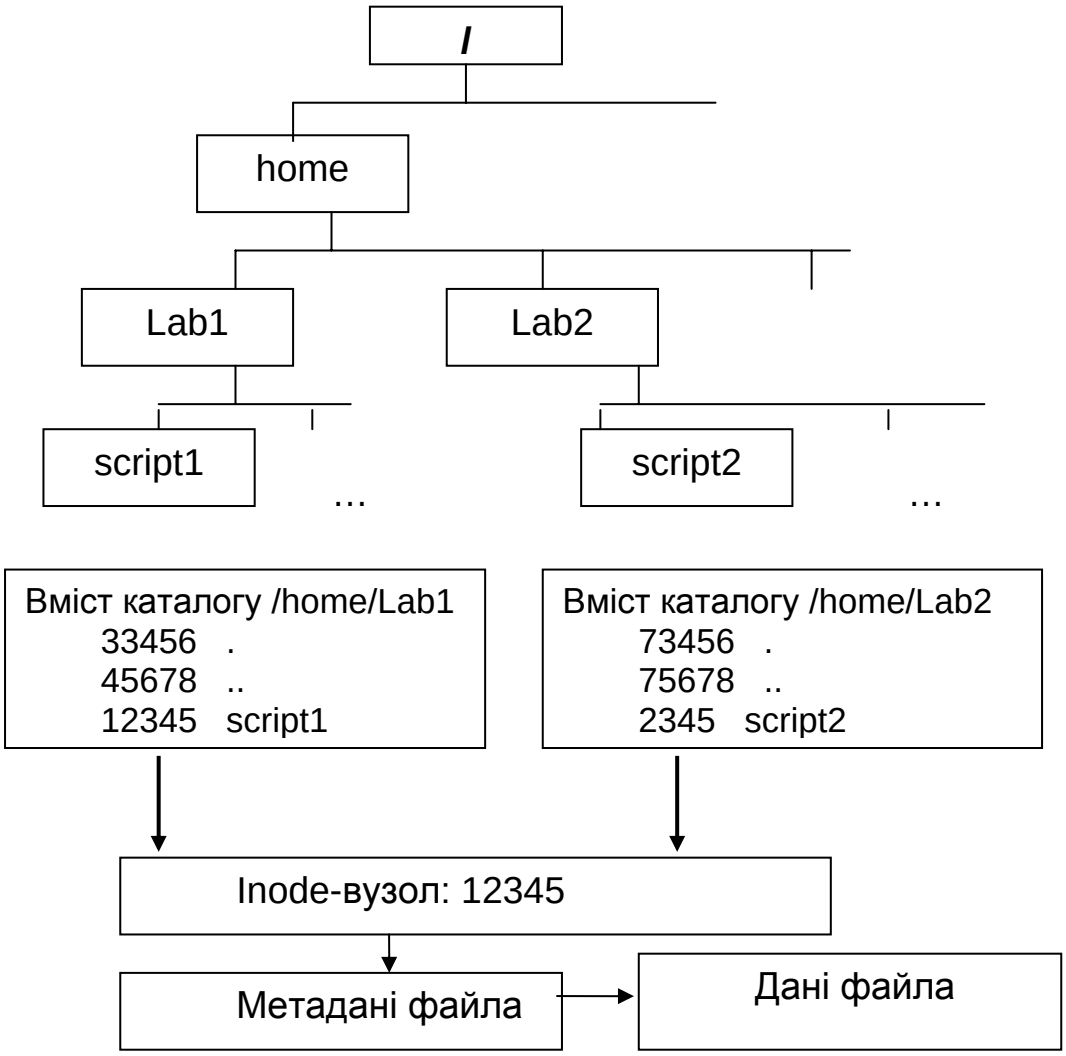


Рис. 3.2. Жорсткий зв'язок імен з даними файла

Він означає, що у відповідному каталозі імені зв'язку зіставляється ім'я inode-вузла відповідного файла. Жорсткі зв'язки абсолютно рівноправні. Нові жорсткі зв'язки створюються за допомогою команди **`link`** (**`ln`**). Наприклад, наведена нижче команда **`ln`** дозволяє створи-

ти ще одне ім'я (**script2**) для файла **script1**; при цьому створюється новий елемент каталогу з тим же номером inode-вузла, що належить файлу **script1**:

ln script1 /home/Lab2/script2.

У списках файлів у каталогах файли жорстого зв'язку для одного файла відрізняються тільки іменами.

Інформацію про наявність у файла декількох імен, зв'язаних з ним жорсткими зв'язками, можна одержати з повного лістингу файлів каталогу (друге слово в дескрипторі файла).

Імена всіх зв'язків даного файла можна довідатися за допомогою команди **ncheck**, якщо вказати в числі її параметрів номер inode-вузла відповідного файла.

Символічний зв'язок є особливим типом файла, що створюється за допомогою команди **link** із ключем **l**. При виконанні цієї команди у відповідному каталозі створюється спеціальний файл, у якому зберігається ім'я файла, на який символічний зв'язок посилається. Для символічного зв'язку створюється окремий inode-вузол і виділяється окремий блок даних для збереження потенційно довгого імені файла. При виведенні на екран файла символічного зв'язку операційна система виводить дані основного файла. Організація символічного зв'язку показана на рис. 3.3.

3.4. Атрибути файлів

Оскільки ОС UNIX є мультикористувальницькою операційною системою, у ній організована авторизація доступу різних користувачів до файлів файлової системи. Під авторизацією доступу розуміються дії системи, що допускають чи не допускають доступ даного користувача до даного файла залежно від прав доступу користувача, установлених для файла. Схема авторизації доступу, застосована в ОС UNIX, стала фактичним стандартом сучасних операційних систем. Вона основана на використанні ідентифікаторів користувача і групи користувачів.

В ОС UNIX існують три базових класи доступу, для кожного з яких можливе призначення відповідних прав доступу:

- u (user access) – права доступу власника файла;
- g (group access) – права доступу групи, до складу якої входить власник файла;
- o (other access) – права доступу інших користувачів (за винятком суперкористувача, права доступу якого не обмежені).

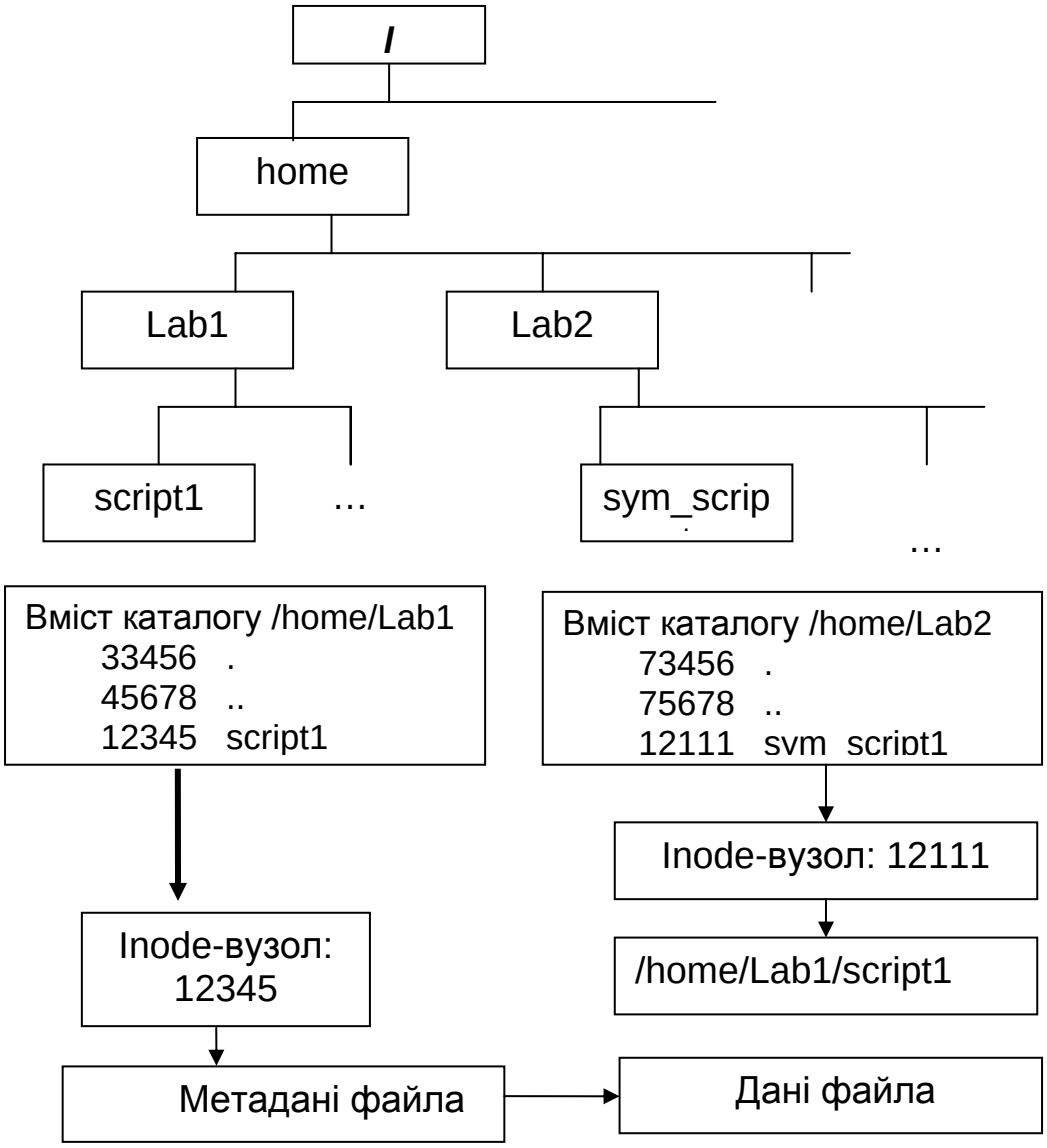
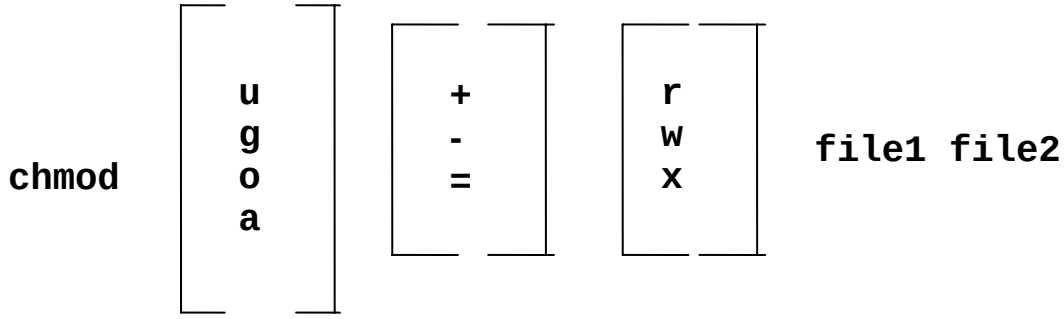


Рис. 3.3. Організація символічного зв'язку в ОС UNIX

В ОС UNIX існують три типи прав доступу для кожного класу: r (read) – право на читання; w (write) – право на запис, тобто модифікацію; x (execute) – право виконання.

Права доступу для кожного файла відображаються при повному лістингу каталогу в першій колонці трьома тріадами, починаючи з другого символу колонки (перший символ вказує тип файла). Наявність права доступу позначається відповідним символом, а його відсутність – символом дефіс.

Права доступу можуть змінюватися тільки власником файла чи суперкористувачем за допомогою команди **chmod**. Формат цієї команди має такий вигляд:



Перший аргумент указує клас доступу (**u** – власник-користувач, **g** – власник-група, **o** – інші користувачі, **a** – усі класи користувачів), третій – права доступу (**r** – читання, **w** – запис, **x** – виконання), другий – операцію, яку необхідно виконати (**+** – додати, **-** – видалити, **=** – надати) для списку файлів **file1**, **file2** і т.д.

Наприклад:

chmod a+w f1 – надання права запису у файл **f1** для всіх користувачів;

chmod go=r f1 – установлює для файла **f1** право читання для всіх користувачів, крім власника;

chmod g+x-w f1 – додає для групи право на виконання файла **f1** і знімає право на запис;

chmode u=rwx, go=rx f1 – перетворює файл **f1** у скрипт, дозволяючи для всіх запуск його на виконання.

Тріади прав доступу в цій команді можна задавати у восьмеричній системі (право є – 1, права немає – 0). Тоді остання команда може бути записана так: **chmode 755 file1**

3.4.1. Ідентифікатори користувача і групи користувачів

З кожним процесом, який виконується, в ОС UNIX зв'язуються такі ідентифікатори користувача: реальний (real user ID), ефективний (effective user ID) і збережений (saved user ID). Аналогічно, з кожним процесом зв'язуються три ідентифікатори групи користувачів - real group ID, effective group ID і saved group ID.

При вході користувача в систему програма **login** перевіряє, що користувач зареєстрований у системі й знає правильний пароль (якщо він установлений), утворює новий процес і запускає в ньому необхідний для користувача інтерпретатор shell. Але перед цим **login** установлює для створеного процесу ідентифікатори користувача і групи, використовуючи для цього інформацію, що зберігається у файлах **/etc/passwd** і **/etc/group**. Після того, як із процесом зв'язуються

ідентифікатори користувача і групи, для цього процесу починають діяти обмеження для доступу до файлів. Процес може одержати доступ до файла чи виконати його (якщо файл містить програмний код) тільки в тому випадку, якщо обмеження доступу, що зберігаються при файлі, дозволяють це зробити. Зв'язані з процесом ідентифікатори передаються створюваним їм процесам, поширюючи на них ті ж самі обмеження. Однак у деяких випадках процес може змінити свої права, а іноді система може змінити права доступу процесу автоматично.

Розглянемо приклад. У файл **/etc/passwd** заборонений запис усім, крім суперкористувача. Цей файл крім іншого містить паролі користувачів, і кожному користувачеві дозволяється змінювати свій пароль. Є спеціальна програма **/bin/passwd**, що змінює паролі. Однак користувач не може змінити пароль навіть за допомогою цієї програми, оскільки запис у файл **/etc/passwd** для нього заборонений. У системі UNIX ця проблема вирішується так. При запуску програми для відповідного процесу можливо установити ідентифікатор користувача, що відповідає не ідентифікатору користувача і/чи групи, а ідентифікатору власника виконуваного файла. Зокрема, при запуску програми **/bin/passwd** процес одержить ідентифікатор суперкористувача, і програма зможе зробити запис у файл /**etc/passwd**.

І для ідентифікатора користувача, і для ідентифікатора групи реальний ID є дійсним ідентифікатором, а ефективний ID – ідентифікатором поточного виконання. В ОС UNIX існують додаткові атрибути, за допомогою яких можна змінити стандартне виконання деяких операцій. Ці атрибути встановлюються утилітою **chmode** з використанням спеціальних опцій. У табл. 3.3 подано додаткові атрибути для звичайних файлів і для каталогів.

Таблиця 3.3

Додаткові атрибути для звичайних файлів і каталогів в ОС UNIX

Опція для команди chmode	*Назва атрибута*	*Призначення атрибута*
Додаткові атрибути для звичайних файлів		
s	Set UID,SUID	Установити при виконанні UID процесу
s	Set GID,GUID	Установити при виконанні GID процесу
t	Блокування	Установити обов'язкове блокування файлів
Додаткові атрибути для каталогів		
s	Sticky bit	Можливість видаляти власні файли та файли, що дозволяється модифікувати
t	Set GID,GUID	Дозвіл змінити правило установлення власника-групи

3.4.2. Захист файлів

Як і прийнято в мультикористувальницькій операційній системі, в ОС UNIX підтримується однаковий механізм контролю доступу до файлів і каталогів файлової системи. Будь-який процес може одержати доступ до файла в тому і тільки в тому випадку, якщо права доступу, описані при файлі, відповідають можливостям цього процесу.

Захист файлів від несанкціонованого доступу в ОС UNIX ґрунтується на трьох фактах. По-перше, з будь-яким процесом, що створює файл (чи каталог), асоційований унікальний у системі ідентифікатор користувача (UID – User Identifier), що надалі можна трактувати як ідентифікатор власника знову створеного файла. По-друге, з кожним процесом, що намагається одержати доступ до файла, зв'язана пара ідентифікаторів – поточні ідентифікатори користувача і його груп. По-третє, кожному файлу однозначно відповідає його індексний дескриптор – inode-вузол.

В ОС UNIX імена та вміст файлів не зв'язані однозначно: при наявності декількох жорстких зв'язків з одним файлом кілька імен файла реально вказують на той самий файл і асоційовані з тим самим inode-вузлом. Серед інформації, що міститься в індексному дескрипторі, знаходяться дані, що дозволяють файловій системі оцінити права доступу цього процесу до даного файла в необхідному режимі.

Загальні принципи захисту однакові для всіх існуючих варіантів системи: інформація inode-вузла містить UID і GID поточного власника файла. Крім того, в inode-вузлі файла зберігається шкала, у якій вказано, що може робити з файлом користувач-власник, що можуть робити з файлом користувачі, що входять у групу користувачів, і що можуть робити з файлом інші користувачі.

3.5. Основні утиліти для роботи з файлами

ОС UNIX має багату бібліотеку утиліт, що дозволяють працювати з файловою системою і доступні як команди командного інтерпретатора. У табл. 3.4 подано найбільш уживані з них. Докладнішу інформацію про утиліти ОС UNIX можна знайти в електронному довіднику man.

Таблиця 3.4

Основні команди інтерпретатора bash для роботи з файлами

Назва утиліти	*Призначення утиліти*
cp file1 file2	Копіювання файла **file1** у файл **file2**
rm file1	Анулювання файла **file1**
mv file1 file2	Перейменування файла **file1** у файл **file2**
mkdir file	Створення нового каталогу **file**
rmdir file	Анулювання каталогу **file**
ls file	Видача вмісту каталогу **file**
Pwd	Видача повного імені поточного каталогу
cat file	Видача на екран умісту файла **file**
chown file режим	Зміна режиму доступу до файла

При вході в систему користувач опиняється у визначеній заздалегідь вершині дерева. Нехай, наприклад, це буде каталог **/usr**.

Змінити місцезнаходження можна командою **cd <каталог>**. Так перейти в каталог /**usr/include/sys** можна за допомогою команди **cd /usr/include/sys** (тут зазначене повне ім'я каталогу), чи за допомогою команди **cd include/sys** (тут зазначене відносне ім'я каталогу).

Ознака відносного імені – відсутність символу **/** на початку імені.

Команда **cd ..** здійснить перехід нагору на батьківський рівень. З каталогу **/usr/include/sys** відбудеться перехід у каталог **/usr/include**, а команда **cd** (без параметрів) здійснить перехід у початковий директорій користувача (тобто директорій, у якому користувач опиняється при вході в систему).

Створити нові каталоги можна за допомогою такої команди:

mkdir <імена створюваних каталогів>.

Так команда **mkdir err new** створить у даному каталозі два нових каталоги з відносними іменами **err** і **new**.

Анулювати порожній каталог можна за допомогою команди **rmdir <імена каталогів, що видаляються,>**.

Найбільш природний для користувача спосіб створення файлів – це використання будь-якого текстового чи екранного редактора.

Командою **rm file1** можна видалити **file1**.

Видалити групу файлів можна такою командою: **rm <імена файлів, що видаляються>**.

Командою **mv стар-ім'я нов-ім'я** можна перейменувати файл.

Командою **cp стар-ім'я нов-ім'я** можна скопіювати файл (зберігши також старий).

3.6. Висновки

Розгляд цієї теми включає:

1) структуру файлової системи ОС UNIX;

2) типи й атрибути файлів;

3) основи базової файлової системи System V.

4. ПІДСИСТЕМА КЕРУВАННЯ ПРОЦЕСАМИ

4.1 Структура процесу в ОС UNIX

Процес – це блок програми, що диспетчиризується; він складається з послідовності машинних інструкцій програми, даних і стекових структур, причому кілька процесів можуть належати одній програмі. Дія процесу полягає у виконанні його набору інструкцій, що є замкнутим і не передає керування набору інструкцій іншого процесу. Він зчитує і записує інформацію в розділ даних і в стек, але йому недоступні дані та стеки інших процесів. Процеси взаємодіють один з одним за допомогою спеціальних засобів взаємодії, наданих операційною системою.

Кожен процес, за винятком нульового, в ОС UNIX породжується в результаті запуску іншим процесом системної операції **fork**. Процес, що запустив операцію fork, називається батьківським, а процес, знову створений, – породженим або потомком. Кожен процес має одного батька, але може породити багато процесів. Ядро системи ідентифікує кожен процес за його номером, що називається ідентифікатором процесу (PID). Нульовий процес є особливим процесом, який створюється в процесі завантаження системи; цей процес, що має ім'я **init**, є предком будь-якого іншого процесу в системі.

Користувач, транслюючи вихідний текст програми, створює файл, який складається з декількох частин:

- секції заголовка, що описує атрибути файла;
- тексту програми;
- даних, які мають початкові значення при запуску програми на виконання, і даних, не ініціалізованих у момент запуску;

- інших секцій, наприклад коду бібліотечних функцій.

При запуску файла, що виконується, ядро створює процес, якому виділяються три інформаційні структури (рис. 4.1): адресний простір у режимі задачі, адресний простір у режимі ядра, рядок у системній таблиці процесів.

Адресний простір процесу в режимі задачі складається з трьох основних областей: інструкцій, даних і стека. Області інструкцій і даних відповідають однойменним секціям файла, а область стека створюється автоматично, її розмір динамічно встановлюється ядром системи під час виконання.

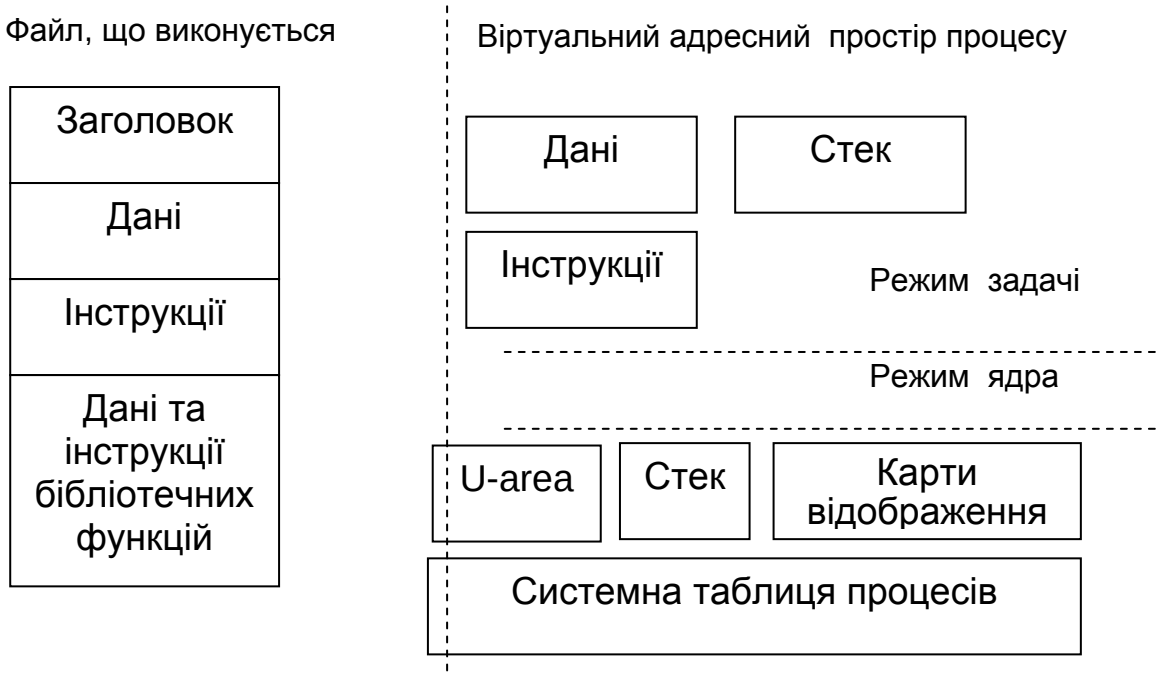


Рис. 4.1. Структура процесу ОС UNIX

Дві приналежні ядру структури даних описують процес: запис у таблиці процесів і простір процесу в режимі ядра; ці структури містять керуючу інформацію і дані про стан процесу.

Таблиця процесів містить поля, які мають бути завжди доступні ядру, а простір процесу – поля, необхідність у яких виникає тільки у процесу, що виконується. Тому ядро виділяє місце для простору процесу тільки при створенні процесу: у ньому немає необхідності, якщо запису в таблиці процесів не відповідає конкретний процес.

Запис у таблиці процесів складається з таких полів:

- поле стану, що ідентифікує стан процесу;

- поля, які використовуються ядром при розміщенні процесу і його простору в основній чи зовнішній пам'яті. Ядро використовує інформацію цих полів для переключення контексту на процес, коли процес

переходить зі стану готовності до виконання в стан виконання в режимі ядра або зі стану резервування в стан виконання в режимі задачі. Крім того, ядро використовує цю інформацію при перекачуванні процесів з/в оперативну пам'ять. Запис у таблиці процесів містить також поле, що описує розмір процесу і дозволяє ядру планувати виділення простору для процесу;

- кілька ідентифікаторів користувача, що встановлюють різні привілеї процесу;

- ідентифікатори процесу (PID), що вказують на взаємозв'язок між процесами (значення полів PID задаються при переході процесу в стан "створений" під час виконання функції **`fork`**);

- дескриптор події (установлюється при припиненні процесу);

- параметри планування, що дозволяють ядру встановлювати порядок переходу процесів зі стану виконання в режимі ядра в стан виконання в режимі задачі;

- поле сигналів, у якому перелічуються сигнали, послані процесу, але ще не оброблені;

- різні таймери, які описують час виконання процесу і використання ресурсів ядра та дозволяють здійснювати спостереження за виконанням і обчисленням динамічного пріоритету; одне з полів є таймером, що встановлює користувач; він необхідний для посилки процесу сигналу тривоги.

Адресний простір процесу в режимі ядра включає структуру u-area, яка є розширенням відповідного запису в таблиці процесів і містить інформацію, що описує процес; доступ до цієї інформації має забезпечуватися тільки під час виконання процесу. Важливими елементами цієї інформації є:

- покажчик на позицію в таблиці процесів, що відповідає поточному процесу;

- параметри поточної системної операції, які повертають значення і коди помилок;

- дескриптори для усіх відкритих файлів;

- внутрішні параметри введення-виведення;

- поточний каталог і поточний корінь;

- межі файлів і процесу.

Ядро системи має безпосередній доступ до полів адресного простору задачі, який виділений процесу, що виконується, але не має доступу до відповідних полів інших процесів. При звертанні до адресного простору задачі, який виділений активному процесу, ядро посилається на структурну змінну u-area. Коли запускається на виконання інший процес, ядро перенастроює віртуальні адреси таким чином,

щоб структурна перемінна u-area вказувала на адресний простір задачі для нового процесу. Процес у системі UNIX може виконуватися в двох режимах: режимі ядра чи режимі задачі; у кожнім з цих режимів процес користується окремим стеком. *Стек задачі* містить аргументи, локальні змінні й іншу інформацію щодо функцій, які виконуються в режимі задачі. *Стек ядра* містить записи активації для функцій, що виконуються в режимі ядра. Елементи функцій і даних у стеці ядра відповідають функціям і даним, що відносяться до ядра, але не до програми користувача, проте конструкція стека ядра подібна до конструкції стека задачі. Стек ядра для процесу порожній, якщо процес виконується в режимі задачі.

4.2. Типи й атрибути процесів

В ОС UNIX підтримуються такі *типи процесів*:

- системні процеси, які є частиною ядра, завжди розташовані в оперативній пам'яті та мають можливості звертатися до даних, недоступних іншим процесам; ці процеси не мають відповідних їм файлів, їх програмний код є частиною програмного коду ядра;
- демони – неінтерактивні процеси, що запускаються шляхом завантаження в пам'ять відповідних їм файлів і забезпечують роботу різних підсистем ОС UNIX (системи друку, мережевих послуг, термінального доступу і т.д.);
- прикладні процеси – всі інші процеси, що виконуються в системі в рамках сеансу роботи користувача.

Для керування процесами в ОС UNIX процеси описуються набором атрибутів, основні з яких подано в табл. 4.1.

Таблиця 4.1

Атрибути процесів

Атрибут	*Призначення атрибута*
PID (Process ID)	Унікальний числовий ідентифікатор процесу (надається процесу при його створенні, звільняється після завершення процесу)
PPID (Parent Process ID)	Ідентифікатор батьківського процесу
Nice Number	Відносний пріоритет процесу, що не змінюється протягом всього життя процесу (може бути змінений командним шляхом користувачем чи адміністратором системи; використовується при обчисленні динамічного пріоритету процесу, що є основою роботи планувальника процесів)

Закінчення таблиці 4.1

Атрибут	*Призначення атрибута*
TTY	Термінал, асоційований із процесом
RUID (Real User ID)	Реальний ідентифікатор, що збігається з ідентифікатором користувача, який запустив процес
EUID (Effective User ID)	Ефективний ідентифікатор, що зазвичай збігається з реальним. Існує можливість змінити ефективний ідентифікатор (установленням спеціального прапорця), присвоїв йому значення ідентифікатора власника файла, який виконується; це дозволяє розширити права доступу процесу до ресурсів
RGID (Real Group ID)	Реальний ідентифікатор групи дорівнює ідентифікатору групи користувача, який запустив процес
EGID (Effective Group ID)	Аналогічний EUID, але дозволяє змінювати клас доступу для групи користувачів

4.3. Стан процесу і діаграма переходів зі стану в стан

Єдиним засобом створення користувачем нового процесу в операційній системі UNIX є виконання системної функції **`fork`**. Процес, що викликає функцію **`fork`**, називається батьківським (процес-предок); процес, який створюється, називається породженим (процес-нащадок). Синтаксис виклику функції **`fork`**: **`pid = fork().`**

У ході реалізації функції **`fork`** ядро виконує такі дії:

- виділяє пам'ять під описувач нового процесу в таблиці описувачів процесів;
- призначає унікальний ідентифікатор процесу (PID) для нового процесу;
- створює логічну копію процесу, що виконує системний виклик fork, у тому числі повне копіювання вмісту віртуальної пам'яті процесу-предка в створювану віртуальну пам'ять, а також копіювання складових ядерного статичного і динамічного контекстів процесу-предка;
- збільшує лічильники відкриття файлів (процес-нащадок автоматично успадковує усі відкриті файли свого предка);
- повертає знову утворений ідентифікатор процесу в точку повернення із системного виклику в процесі-предку і повертає значення 0 у точці повернення в процесі-нащадку.

На рис 4.2 наведено алгоритм створення процесу. Спочатку ядро має упевнитися в тому, що для успішного виконання алгоритму **fork** є всі необхідні ресурси. Якщо вільних ресурсів немає, функція **fork** завершується невдало. Ядро шукає місце в таблиці процесів для конструювання контексту породжуваного процесу і перевіряє, чи не перевищив користувач, що виконує **fork**, обмеження на максимально припустиму кількість паралельно запущених процесів. Ядро також назначає для нового процесу унікальний ідентифікатор, значення якого перевищує на одиницю максимальний з існуючих ідентифікаторів. Як тільки буде досягнуто максимально припустиме значення, відлік ідентифікаторів знову починається з нуля. Оскільки більшість процесів має короткий час життя, при переході до початку відліку значна частина ідентифікаторів виявляється вільною.

У результаті виконання функції **fork** користувальницький контекст процесу-предка і процесу-нащадка збігається в усьому, окрім значення, що повертається, і змінної PID.

На кількість процесів, які одночасно виконуються, накладається обмеження: жоден з користувачів не може займати в таблиці процесів занадто багато місця, заважаючи тим самим іншим користувачам створювати нові процеси. Крім того, простим користувачам не дозволяється створювати процес, що займає останнє вільне місце в таблиці процесів, у противному випадку система зайшла б у тупик, оскільки якщо в таблиці процесів немає вільного місця, то ядро не може гарантувати, що всі існуючі процеси завершаться природним чином. Суперкористувачу операційна система дає можливість виконувати стільки процесів, скільки йому буде потрібно, звичайно, з огляду на розмір таблиці процесів; при цьому процес, що виконується суперкористувачем, може зайняти в таблиці й останнє вільне місце. Передбачається, що суперкористувач може запускати процес, що спонукає інші процеси до завершення, якщо це необхідно.

Ядро надає початкові значення різним полям запису таблиці процесів (що відповідає породженому процесу), копіюючи в них значення полів із запису батьківського процесу. Наприклад, породжений процес успадковує з батьківського процесу коди ідентифікації користувача (реальний код і той, під яким виконується процес), групу процесів, керовану батьківським процесом, а також значення рівня статичного пріоритету, яке задане батьківським процесом у функції **nice** і використовується при обчисленні пріоритету планування.

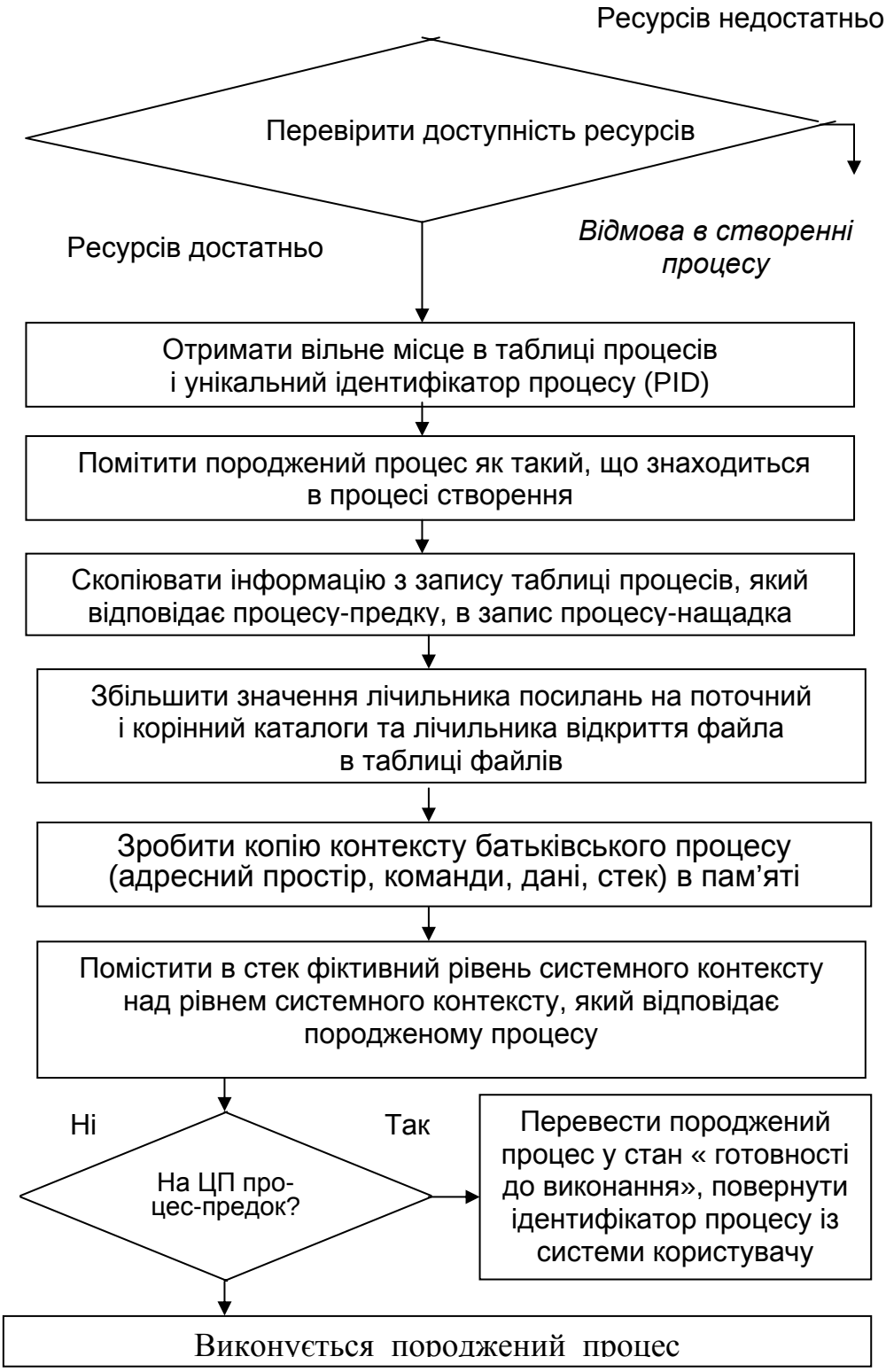


Рис. 4.2. Алгоритм створення процесу

Ядро передає значення поля ідентифікатора батьківського процесу в запис породженого, включаючи останній у деревоподібну структуру процесів, і присвоює початкові значення різним параметрам

планування, таким, як пріоритет планування, використання ресурсів центрального процесора й інші значення полів синхронізації. Потім ядро встановлює значення лічильників посилань на файли, з якими автоматично зв'язується породжуваний процес. Породжений процес розміщається в поточному каталозі батьківського процесу. Число процесів, що звертаються в даний момент до каталогу, збільшується на 1, і, відповідно, збільшується значення лічильника посилань на його індекс. Якщо батьківський процес або один з його предків уже змінював кореневий каталог, породжений процес успадковує і новий корінь з відповідним збільшенням значення лічильника посилань на індекс кореня. Нарешті, ядро виконує перегляд таблиці дескрипторів користувача для батьківського процесу в пошуках відкритих файлів, які відомі процесу, і збільшує значення лічильника посилань, асоційованого з кожним з відкритих файлів, у глобальній таблиці файлів. Породжений процес не лише успадковує права доступу до відкритих файлів, але і поділяє доступ до файлів з батьківським процесом, оскільки обидва процеси звертаються в таблиці файлів до тих самим записів.

Після завершення цих дій ядро готове до створення користувальницького контексту для породженого процесу . Ядро виділяє пам'ять для адресного простору процесу, його областей і таблиць сторінок, створює копії всіх областей батьківського процесу і приєднує кожну область до породженого процесу. Вміст адресного простору породженого процесу збігається з вмістом простору батьківського процесу. Після створення процесу предок і нащадок починають своє власне життя, довільно змінюючи свій контекст. Після завершення створення породженого процесу батьківський процес може відкрити новий файл, до якого породжений процес уже не одержить доступу автоматично.

Батьківський і породжений процеси спільно користуються файлами, що були відкриті батьківським процесом до моменту виконання функції **`fork`**, при цьому значення лічильника посилань на кожний з цих файлів у таблиці файлів на одиницю більше, ніж до виклику функції. Породжений процес має ті ж, що і батьківський процес, поточний і кореневий каталоги. Значення ж лічильника посилань на індекс кожного з цих каталогів так само стає на одиницю більше, ніж до виклику функції. Вміст областей команд, даних і стека (задачі) обох процесів збігається.

Час життя процесу складається з декількох станів, кожний з яких має характеристики, що описують процес. З моменту створення до моменту завершення процес може знаходитися в одному з таких станів (рис. 4.3):

1. Процес виконується в режимі задачі, тобто центральний процесор виконує інструкції цього процесу.

2. Процес виконується в режимі ядра, тобто центральний процесор виконує системні інструкції ядра від імені цього процесу.

3. Процес не виконується, але готовий до виконання під керуванням ядра і чекає, коли планувальник вибере його; у цьому стані може знаходитися багато процесів, і алгоритм планування встановлює, який із процесів буде виконуватися наступним.

4. Процес припинений і знаходиться в оперативній пам'яті.

5. Процес готовий до запуску, але програма підкачування (нульовий процес) має ще завантажити цей процес в оперативну пам'ять, перш ніж він буде запущений під керуванням ядра.

6. Процес припинений і програма підкачування вивантажила його в зовнішню пам'ять, щоб в оперативній пам'яті звільнити місце для інших процесів.

7. Процес повернутий із привілейованого режиму (режиму ядра) у непривілейований (режим задачі), ядро резервує його і переключає контекст на інший процес.

8. Процес знову створений і знаходиться в перехідному стані; процес існує, але не готовий до виконання, хоча і не припинений; цей стан є початковим станом усіх процесів, крім нульового.

9. Процес припиняє існування, але в системі існує його батьківський процес; даний процес переходить у стан зомбі, звільняючи усі надані йому ресурси, крім рядка в системній таблиці процесів, що містить код виходу і деяку хронометричну статистику для батьківського процесу.

Оскільки на процесорі в кожен момент часу виконується тільки один процес, у станах 1 і 2 може знаходитися тільки один процес. Ці два стани відповідають двом режимам виконання – режиму задачі й режиму ядра.

Початковим станом процесу є створення його батьківським процесом (за допомогою системної функції **`fork`**). З цього стану процес переходить у стан готовності до запуску (3 або 4). Якщо процес знаходиться в стані готовності до запуску в пам'яті (3), то планувальник процесів раніше чи пізніше вибере процес для виконання, і процес перейде в стан виконання в режимі ядра. Потім цей процес може перейти в стан виконання в режимі задачі. У цей час може відбутися переривання роботи процесора за таймером, і процес знову перейде в стан виконання в режимі ядра. Як тільки програма обробки переривання закінчить роботу, ядро може запустити інший (більш пріоритетний) процес, тоді перший процес перейде в стан резервування, звільнивши центральний процесор іншому процесу.

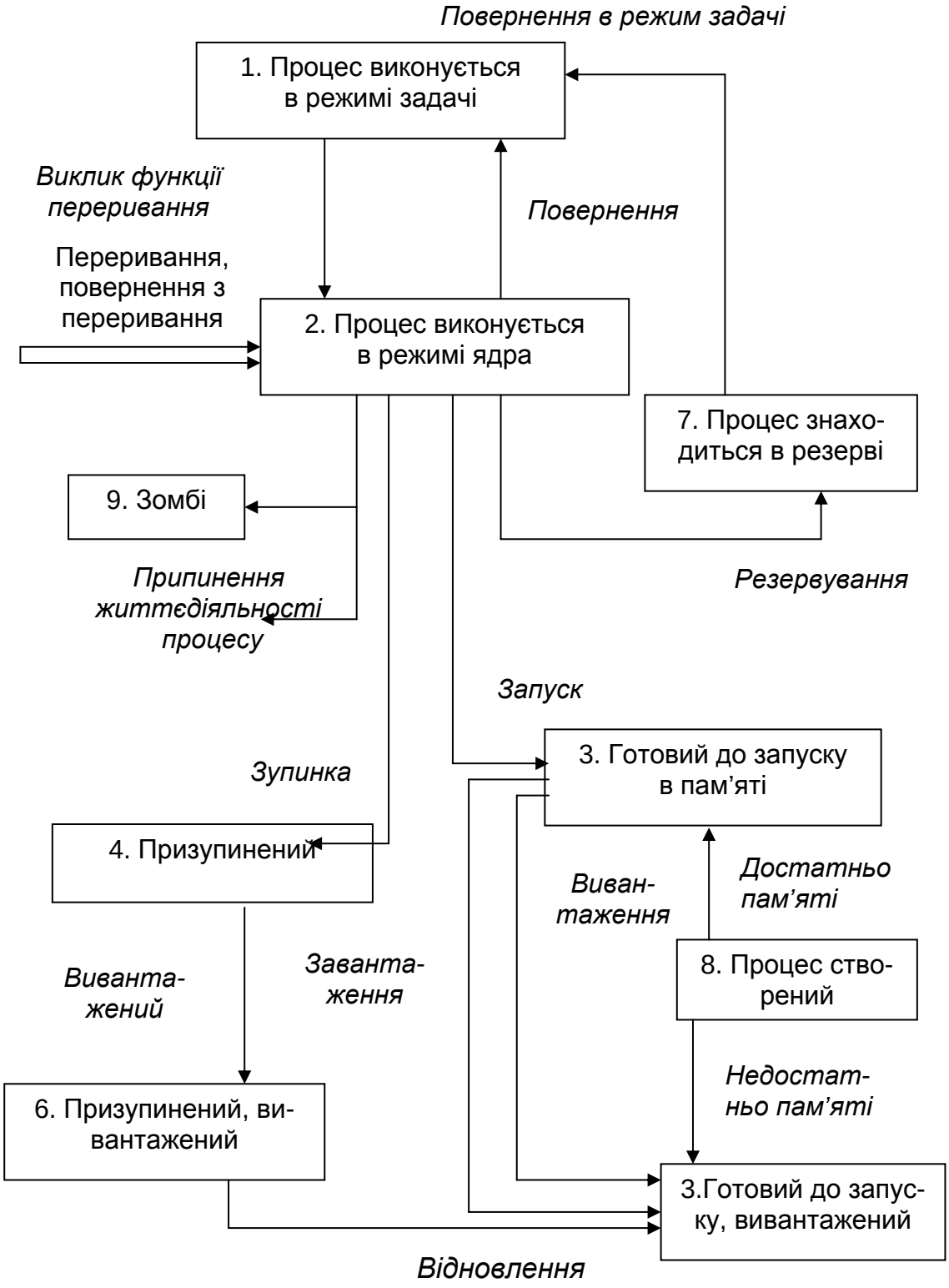


Рис. 4.3. Стани процесу і діаграма переходу процесу зі стану в стан

Стан резервування фактично не відрізняється від стану готовності до запуску в пам'яті, але він виділяється в окремий стан, щоб підкреслити, що процес, який виконується в режимі ядра, може бути зарезервований тільки в тому випадку, якщо він збирається повернутися в режим задачі. Отже, ядро може за необхідності підкачувати процес зі стану резервування. Коли планувальник знову вибере процес для виконання, той знову повернеться в стан виконання в режимі задачі.

Коли процес викликає системну функцію, він зі стану виконання в режимі задачі переходить у стан виконання в режимі ядра. Припустимо, що системній функції потрібно введення-виведення з диска і тому процес змушений чекати завершення введення-виведення. Він переходить у стан призупинення в пам'яті, у якому буде знаходитися доти, поки не одержить повідомлення про закінчення введення-виведення. Коли ці операції закінчаться, відбудеться апаратне переривання роботи центрального процесора і програма обробки переривання відновить виконання процесу, у результаті чого він перейде в стан готовності до запуску в пам'яті.

Якщо система виконує таку кількість процесів, які одночасно не можуть розміститися в оперативній пам'яті, програма підкачування (нульовий процес) вивантажує один процес, щоб звільнити місце для іншого процесу, що знаходиться в стані готовності до запуску, але є вивантаженим з оперативної пам'яті. Процес, що завантажується, переходить у стан готовності до запуску в пам'яті, а що вивантажується – у стан готовності до запуску, але вивантаженого з оперативної пам'яті.

Коли планувальник вибирає процес для виконання, він переходить у стан виконання в режимі ядра. Коли процес завершується, він переходить зі стану виконання в режимі ядра в стан припинення існування.

Процес може керувати деякими з переходів на рівні задачі. Але в який зі станів процес перейде після створення (тобто в стан "готовий до виконання, знаходячись у пам'яті", чи в стан "готовий до виконання, але вивантажений"), залежить від ядра. Процесу ці стани не підконтрольні. Під час роботи процес може звертатися до різних системних функцій, щоб перейти зі стану виконання в режимі задачі в стан виконання в режимі ядра. Але момент повернення з режиму ядра від процесу вже не залежить; у результаті якихось подій він може ніколи не повернутися з цього режиму і з нього перейти в стан припинення існування. Але процес може завершитися і за своєю власною волею (за допомогою функції **`exit`**).

Процеси можуть припиняти своє виконання, якщо вони очікують виникнення деякої події, наприклад завершення введення-виведення на периферійному пристрої, виходу, виділення системних ресурсів і т.д. При цьому процес переходить у стан сну до настання очікуваної події, після чого він пробуджується і переходить у стан готовності до виконання. Одночасно можуть припинятися за подією багато процесів; коли подія настає, усі процеси, що припинені за подією, пробуджуються, і планувальник вибирає з них найбільш пріоритетний для подальшої роботи, переводячи його в стан готовності до виконання. Інші процеси знову припиняються, повертаючись у стан сну. Припинені процеси не займають центральний процесор.

Приклад. Нехай три процеси – A, B і C – конкурують за заблокований буфер. Процеси, виявивши, що буфер заблокований, припиняють своє виконання до настання події, за якою буфер буде розблокований. Коли блокування з буфера знімається, всі процеси пробуджуються, переходячи в стан готовності до виконання. Ядро вибирає один із процесів, наприклад B, для виконання. Процес B виявляє, що буфер розблокований, блокує його і продовжує своє виконання. Якщо процес B надалі знову призупиниться без зняття блокування з буфера (наприклад, очікуючи завершення операції введення-виведення), ядро зможе приступити до планування виконання інших процесів. Якщо буде при цьому вибрано процес A, то він знайде, що буфер заблокований, і знову перейде в стан сну; можливо, теж саме відбудеться і з процесом C. Зрештою виконання процесу B відновиться і блокування з буфера буде знято, у результаті чого процеси A і C одержать доступ до нього. Алгоритми переходу в стан сну і пробудження є неподільними. Процес переходить у стан сну миттєво і знаходиться в ньому доти, поки не буде розбуджений.

4.4. Розподіл оперативної пам'яті

Процес у режимі задачі в ОС UNIX складається з трьох логічних секцій: команд, даних і стека. У *секції команд* зберігається набір машинних інструкцій, що виконуються під керуванням процесу; адресами в секції команд виступають адреси команд (для команд переходу і звертань до підпрограм), адреси даних (для звертання до глобальних змінних) і адреси стека (для звертання до структур даних, що локалізовані в підпрограмах). Якщо адреси в коді, що є згенерований, трактувати як адреси у фізичній пам'яті, два процеси не зможуть паралельно виконуватися, якщо їхні адреси перекриваються. Тому компілятор генерує адреси для віртуального адресного простору заданого діапазону, а пристрій керування пам'яттю (диспетчер пам'яті) транслює

віртуальні адреси, які згенеровані компілятором, в адреси комірок, розташованих у фізичній пам'яті. Компілятору немає необхідності знати, у яке місце в пам'яті ядро потім завантажить програму для виконання. В оперативній пам'яті одночасно можуть існувати кілька копій програми: усі вони можуть виконуватися, використовуючи ті самі віртуальні адреси, фактично ж посилаючись на різні фізичні комірки.

4.4.1. Процеси й області пам'яті

Ядро поділяє віртуальний адресний простір процесу на сукупність логічних областей. *Область пам'яті* – це безперервна зона віртуального адресного простору процесу, яка є окремим об'єктом для спільного використання і захисту. Команди, дані та стек однієї програми утворюють автономні області, що належать процесу. Кілька процесів можуть використовувати ту саму область. Наприклад, якщо кілька процесів виконують одну програму, цілком природно, що вони використовують спільну область команд. Аналогічно кілька процесів можуть об'єднатися і використовувати спільну область поділюваної пам'яті.

Ядро підтримує таблицю областей і виділяє запис у таблиці для кожної активної області в системі. Кожен процес має власну таблицю областей процесу. Залежно від версії операційної системи, записи цієї таблиці можуть розташовуватися в таблиці процесів, в адресному просторі процесу чи в окремій області пам'яті. Кожен запис власної таблиці областей процесу містить:

- покажчик на відповідний запис загальної таблиці областей і першу віртуальну адресу процесу в даній області (поділювані області можуть мати різні віртуальні адреси в кожнім процесі);
- поле прав доступу, у якому вказується тип доступу, дозволений процесу: тільки читання, тільки запис чи тільки виконання.

Таблиця областей процесу і структура цієї області аналогічні таблиці файлів і структурі індексу у файловій системі: кілька процесів можуть спільно використовувати адресний простір через область, подібно тому, як вони розділяють доступ до файла за допомогою індексу; кожен процес має доступ до області завдяки використанню запису у власній таблиці областей, так само він звертається до індексу, використовуючи відповідні записи в таблиці користувальницьких дескрипторів файла й у таблиці файлів, що належить ядру.

На рис. 4.4 показано області, власні таблиці областей і віртуальні адреси, у яких ці області з'єднуються, для двох процесів – A і B.

Процеси поділяють область команд AAA за віртуальними адресами 8K і 4K відповідно. Якщо процес A читає комірку пам'яті за адре-

сою 8К, а процес В – комірку за адресою 4К, то вони читають одну і ту саму комірку в області ААА. Області даних і стека у кожного процесу свої.

Поняття області пам'яті не залежить від способу реалізації керування пам'яттю в операційній системі та від способу організації віртуальної пам'яті.

До операцій з керування адресним простором процесу відносяться такі дії: блокування області, зняття блокування з області, виділення області, приєднання області до простору пам'яті процесу, зміна розміру області, завантаження області з файла в простір пам'яті процесу, звільнення області, від'єднання області від простору пам'яті процесу, копіювання вмісту області.

Наприклад, системна функція **exec**, у якій вміст файла, що виконується, накладається на адресний простір задачі, від'єднує використані області, звільняє їх у тому випадку, якщо вони не є поділюваними, виділяє нові області, приєднує їх і завантажує вміст файла.

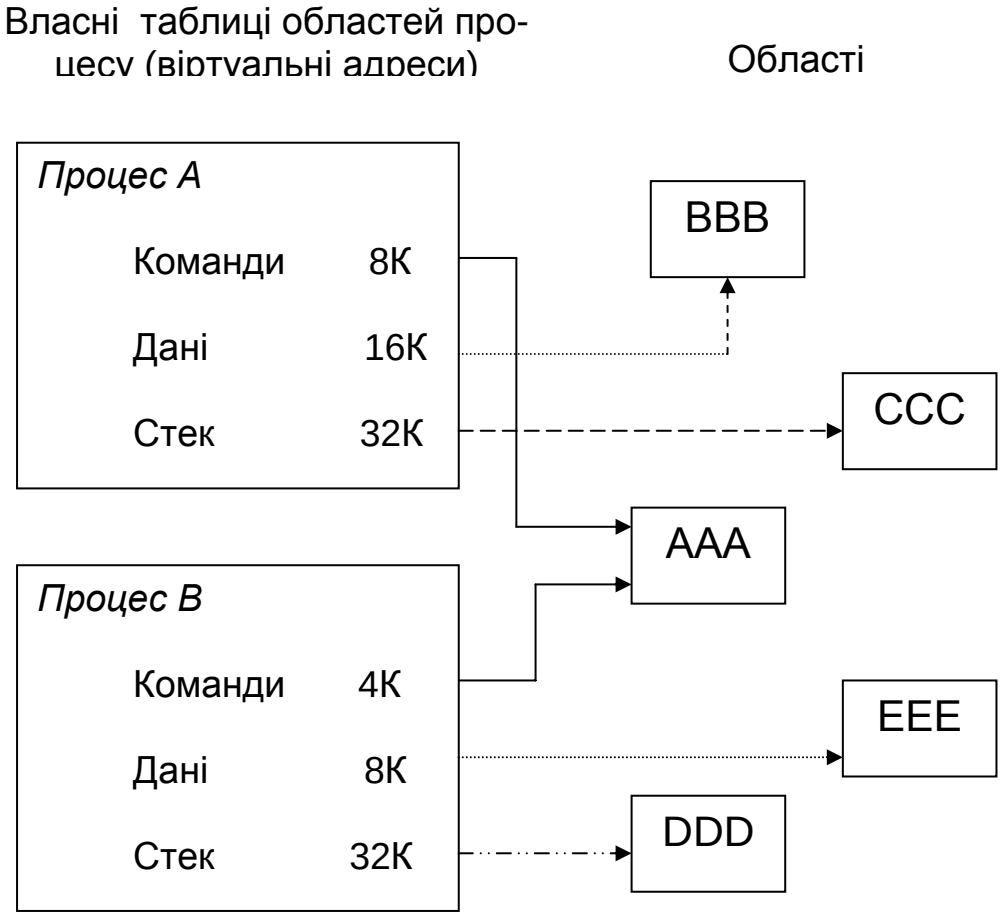


Рис. 4.4. Процеси й області

Усі дії з керування адресним простором процесу виконуються ядром.

4.4.2. Способи організації віртуальної пам'яті

При сторінковій організації віртуальної пам'яті фізична пам'ять поділяється на блоки однакового розміру, які називаються сторінками. Звичайний розмір сторінок становить від 412 байт до 4К і визначається конфігурацією технічних засобів.

Кожна адресна комірка пам'яті міститься в деякій сторінці, і, отже, кожна комірка пам'яті адресується віртуальною адресою, яка складається з двох елементів: номера сторінки і зсуву всередині сторінки в байтах. Наприклад, якщо обсяг машинної пам'яті становить 2^{32} байт, а розмір сторінки – 1К, загальне число сторінок – 22^2 ; можна вважати, що кожна 32-розрядна адреса складається з 22-розрядного номера сторінки і 10-розрядного зсуву всередині сторінки.

Коли ядро призначає області фізичні сторінки пам'яті, не виникає необхідності в призначенні суміжних сторінок і взагалі в дотриманні будь-якої черговості при призначенні.

Метою сторінкової організації пам'яті є підвищення гнучкості призначення фізичної пам'яті, що будується за аналогією з призначенням дискових блоків файлам у файловій системі. Як і при призначенні блоків файлу, так і при призначенні області сторінок пам'яті, основною задачею є підвищення гнучкості та скорочення простору пам'яті, який не використовується.

Ядро встановлює співвідношення між віртуальними адресами області та машинними фізичними адресами за допомогою відображення логічних номерів сторінок в області на фізичні номери сторінок в оперативній пам'яті комп'ютера. Оскільки область – це неперервний простір віртуальних адрес програми, логічний номер сторінки є покажчиком на елемент масиву фізичних номерів сторінок. Запис таблиці областей містить покажчик на таблицю фізичних номерів сторінок, іменовану таблицею сторінок. Записи таблиці сторінок містять машинно-залежну інформацію, таку, як права доступу на читання чи запис сторінки. Ядро підтримує таблиці сторінок у пам'яті та звертається до них так само, як і до всіх інших структур даних ядра.

На рис. 4.5 наведено приклад відображення процесу у фізичні адреси пам'яті.

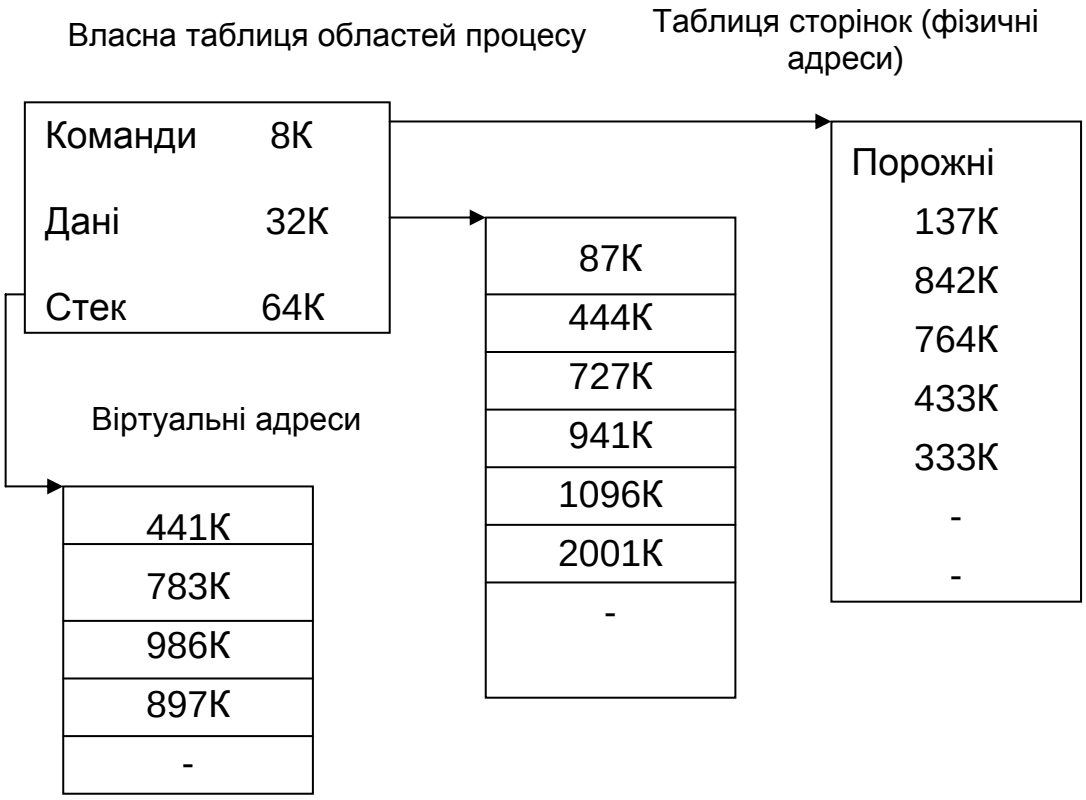


Рис. 4.5. Перетворення віртуальних адрес у фізичні

Нехай розмір сторінки становить 1К і процесу потрібно звернутися до об'єкта в пам'яті, що має віртуальну адресу 68432. З таблиці областей видно, що віртуальна адреса початку області стека – 64436 (64К), якщо припустити, що стек росте в напрямку збільшення адрес. Після віднімання цієї адреси з адреси 68432 одержуємо зсув у байтах усередині області, який дорівнює 2896. Оскільки кожна сторінка має розмір 1К, адреса вказує зі зсувом 848 на другу (починаючи з 0) сторінку області, розташованої за фізичною адресою 986К.

У сучасних комп'ютерах використовуються спеціальні засоби (апаратні регістри і кеші), що підвищують швидкість виконання процедури трансляції адрес. Відновлюючи виконання процесу, ядро за допомогою завантаження відповідних регістрів повідомляє технічним засобам керування пам'яттю, у яких фізичних адресах виконується процес і де розташовуються таблиці сторінок.

При сегментній організації віртуальної пам'яті фізична пам'ять поділяється на блоки різного розміру, які називаються сегментами. Перетворення віртуальних адрес виконується аналогічно перетворенню при використанні сторінкової організації, але в таблиці сегментів для кожного процесу присутня додаткова інформація – розмір кожного сегмента.

При комбінованій організації віртуальної пам'яті фізична пам'ять поділяється на блоки різного розміру (сегменти), які, в свою чергу, поділяються на блоки фіксованого розміру (сторінки).

У сучасних версіях ОС UNIX підтримується сегментна організація пам'яті та комбінована організація з розміром сторінок 4К.

4.4.3. Розміщення ядра

Відображення віртуальних адрес, зв'язаних з ядром, здійснюється незалежно від усіх процесів. Програми та структури даних ядра резидентні в системі й спільно використовуються всіма процесами. При запуску системи відбувається завантаження програм ядра в пам'ять з установленням відповідних таблиць і регістрів для відображення віртуальних адрес ядра у фізичні. Таблиці сторінок для ядра мають структуру, аналогічну структурі таблиці сторінок, зв'язаної з процесом, а механізми відображення віртуальних адрес ядра схожі на механізми, які використовуються для відображення користувальницьких адрес. При роботі в режимі ядра система дозволяє доступ до адрес ядра, при роботі ж у режимі задачі такий доступ заборонений. Наприклад, коли в результаті переривання або виконання системної функції відбувається перехід з режиму задачі в режим ядра, операційна система дозволяє посилання на адреси ядра, а при поверненні в режим задачі ці посилання вже заборонені.

4.4. Контекст процесу

Контекст процесу містить у собі адресний простір задачі, що наданий цьому процесу, а також вміст апаратних регістрів і структур даних ядра.

4.4.1. Компоненти контексту процесу

Контекст процесу поєднує в собі користувальницький, реєстровий і системний контексти.

Користувальницький контекст складається з команд і даних процесу, стека задачі та вмісту спільно використовуваного простору пам'яті у віртуальних адресах процесу. Ті частини віртуального адресного простору процесу, що періодично відсутні в оперативній пам'яті внаслідок вивантаження або заміщення сторінок, також відносяться до користувальницького контексту.

Реєстровий контекст складається з таких компонентів:

- лічильника команд, що вказує адресу тієї команди, яку буде виконувати центральний процесор; ця адреса є віртуальною адресою всередині простору ядра чи простору задачі;

- регістра стану процесора (регістр прапорців), що містить прапорці, які вказують, чи є результат останніх обчислень нульовим, додатним чи від'ємним, чи переповнений регістр з установленням біта переносу і т.д. Операції, що впливають на установку регістра прапорців, виконуються для окремого процесу, тому в цьому регістрі міститься апаратний статус комп'ютера стосовно процесу. Крім того, у регістрі прапорців указується поточний рівень переривання процесора і попередні режими виконання процесу (режим ядра/задачі), а також інформація про те, чи може процес виконувати привілейовані команди і звертатися до адресного простору ядра;

- покажчика вершини стека, у якому міститься адреса наступного елемента стека чи ядра стека задачі, відповідно до режиму виконання процесу;

- регістрів загального призначення, у яких міститься інформація, згенерована процесом під час його виконання.

Системний контекст процесу має статичну частину (перші три елементи в нижченаведеному списку) і динамічну частину (останні два елементи). Протягом усього часу виконання процес постійно використовує одну статичну частину системного контексту, але може мати змінне число динамічних частин. Динамічну частину системного контексту можна подати у вигляді стека, елементами якого є контекстні рівні, що вміщуються в стек чи виштовхуються зі стека ядром при настанні різних подій. Системний контекст містить у собі такі компоненти:

1) запис у таблиці процесів, що описує стан процесу і містить різну керуючу інформацію, до якої ядро завжди може звернутися;

2) частину адресного простору задачі, що виділена процесу для збереження керуючої інформації про процес і доступна тільки в контексті процесу. Загальні керуючі параметри, такі, як пріоритет процесу, зберігаються в таблиці процесів, оскільки звертання до них має виконуватися за межами контексту процесу;

3) записи власної таблиці областей процесу, загальні таблиці областей і таблиці сторінок, необхідні для перетворення віртуальних адрес у фізичні. Якщо кілька процесів спільно використовують загальні області, ці області є складовою частиною контексту кожного процесу, оскільки кожен процес працює з цими областями незалежно від інших процесів. У задачі керування пам'яттю входить ідентифікація

ділянок віртуального адресного простору процесу, що не є резидентними в пам'яті;

4) стек ядра, у якому зберігаються записи процедур ядра, якщо процес виконується в режимі ядра. Незважаючи на те, що всі процеси користуються тими самими програмами ядра, кожний з них має свою власну копію стека ядра для збереження індивідуальних звертань до функцій ядра. Коли процес виконується в режимі задачі, стек ядра порожній;

5) динамічну частину системного контексту процесу, що складається з декількох рівнів і має вид стека, який звільняється від елементів у порядку, зворотному порядку їхнього надходження. На кожнім рівні системного контексту міститься інформація, яка необхідна для відновлення попереднього рівня; вона вміщує в собі реєстровий контекст попереднього рівня.

Ядро розміщує контекстовий рівень у стеку при виникненні переривання, при звертанні до системної функції чи при переключенні контексту процесу. Контекстовий рівень виштовхується зі стека після завершення обробки переривання, при поверненні процесу в режим задачі після виконання системної функції чи при переключенні контексту. Таким чином, переключення контексту спричиняє як вміщення контекстового рівня в стек, так і виштовхування цього рівня зі стека: ядро вміщує в стек контекстовий рівень старого процесу, а виштовхує зі стека контекстовий рівень нового процесу. Інформація, що необхідна для відновлення поточного контекстового рівня, зберігається в запису таблиці процесів.

На рис. 4.6 подано компоненти контексту процесу. У статичну частину контексту входять: користувальницький контекст, що складається з програм процесу (машинних інструкцій), даних, стека і поділюваної пам'яті (якщо вона є), а також статична частина системного контексту, що складається з запису таблиці процесів, простору процесу та записів власної таблиці областей (інформації, необхідної для трансляції віртуальних адрес користувальницького контексту). Динамічна частина контексту має вигляд стека і містить у собі кілька елементів, що зберігають реєстровий контекст попереднього рівня та стек ядра для поточного рівня. Нульовий контекстовий рівень являє собою порожній рівень, що відноситься до користувальницького контексту; збільшення стека тут йде в адресному просторі задачі, стек ядра недійсний. Стрілка, що з'єднує між собою статичну частину системного контексту і верхній рівень динамічної частини контексту, означає те, що в таблиці процесів зберігається інформація, яка дозволяє ядру відновлювати поточний контекстовий рівень процесу.

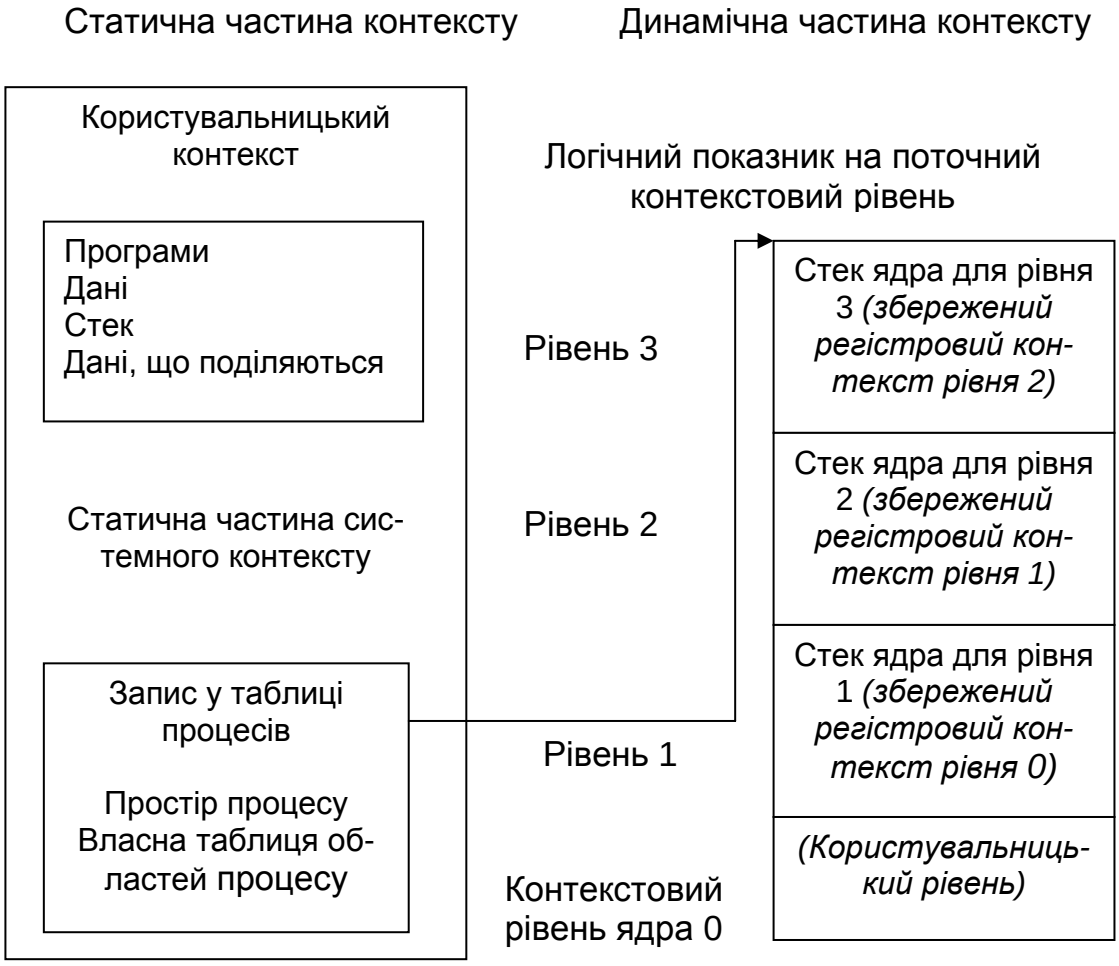


Рис. 4.6. Компоненти контексту процесу

Процес виконується в рамках свого контексту, а точніше, у рамках свого поточного контекстового рівня. Кількість контекстових рівнів обмежується числом наявних у комп'ютера рівнів переривань. Наприклад, якщо в комп'ютері підтримуються чотири рівні переривань (для програм, терміналів, дисків, усіх інших периферійних пристроїв і таймера), то в процесі може бути не більш семи контекстових рівнів: по одному на кожен рівень переривання, один – для системних функцій та один – для користувальницького контексту.

Незважаючи на те, що ядро завжди виконує контекст якого-небудь процесу, логічна функція, яку ядро реалізує в кожен момент, не завжди має відношення до цього процесу. Наприклад, якщо при поверненні даних дисковий запам'ятовуючий пристрій посилає переривання, то переривається виконання поточного процесу і ядро обробляє переривання на новому контекстовому рівні цього процесу, навіть якщо дані відносяться до іншого процесу. Програми обробки пе-

реривань звичайно не звертаються до статичних складових контексту процесу і не видозмінюють їх, тому що ці частини не зв'язані з перериваннями.

4.5. Сигнали як найпростіша форма міжпроцесорної взаємодії

Сигнал -– це засіб інформування процесу з боку ядра про деяку подію, тобто сигнал означає, що обумовлена ним подія відбулася. Але сигнал не несе інформацію про те, скільки однотипних подій відбулося.

Прикладами подій, що генерують сигнали, є:

- закінчення процесу-нащадка;
- виникнення виняткової ситуації в роботі процесу (вихід за припустимі межі віртуальної пам'яті, спроба запису в область віртуальної пам'яті, що доступна тільки для читання і т.д.);
- перевищення верхньої межі системних ресурсів;
- повідомлення про помилки в системних викликах (неіснуючий системний виклик, помилки в параметрах системного виклику, невідповідність системного виклику поточному стану процесу і т.д.);
- сигнали, що посилаються іншим процесам у режимі користувача;
- сигнали, що надходять унаслідок натискання користувачем клавіш на клавіатурі термінала, який зв'язаний з процесом (наприклад, Ctrl-C чи Ctrl-D);
- сигнали, що служать для трасування процесу.

Система надає можливість процесам користувача явно генерувати сигнали, що посилаються іншим процесам. Для цього використовується системний виклик (команда командного інтерпретатора) **`kill(pid, signum)`**. Параметр **`signum`** задає номер сигналу, що генерується (у системному виклику **`kill`** можна вказувати не всі номери сигналів). Параметр **`pid`** має такі значення:

- якщо в значенні **`pid`** зазначене ціле додатне число, то ядро відправить зазначений сигнал процесу, ідентифікатор якого дорівнює **`pid`**;
- якщо значення **`pid`** дорівнює нулю, то зазначений сигнал надсилається всім процесам, які відносяться до тієї ж групи процесів, що і процес, який посилає сигнал (поняття групи процесів аналогічно поняттю групи користувачів; повний ідентифікатор процесу складається з двох частин – ідентифікатора групи процесів та індивідуального ідентифікатора процесу; в одну групу автоматично включаються всі про-

цеси, що мають загального предка; ідентифікатор групи процесу може бути змінений за допомогою системного виклику **setpgrp**);

- якщо значення **pid** дорівнює -1, то ядро надсилає зазначений сигнал усім процесам, дійсний ідентифікатор користувача яких дорівнює ідентифікатору поточного виконання процесу, що посилає сигнал.

Сигнали повідомляють процесам про виникнення асинхронних подій. Сигнали надсилаються процесами (один одному за допомогою функції **kill**) чи ядром. У версії V системи UNIX існують 19 різних сигналів, які можна класифікувати таким чином:

- сигнали, що посилаються у випадку завершення виконання процесу;

- сигнали, що посилаються у випадку виникнення особливих ситуацій, які обумовлені процесом, наприклад звертання до адреси, що знаходиться за межами віртуального адресного простору процесу, спроба запису в область пам'яті, відкриту тільки для читання (наприклад, текст програми), спроба виконання привілейованої команди, виникнення різних апаратних помилок;

- сигнали, що посилаються під час виконання системної функції при виникненні непоправних помилок, таких, як вичерпання системних ресурсів під час виконання функції **exec** після звільнення вихідного адресного простору;

- сигнали, причиною яких служить виникнення під час виконання системної функції зовсім несподіваних помилок, таких, як звертання до неіснуючої системної функції (процес передав номер системної функції, що не відповідає жодній з наявних функцій), запис у канал, не зв'язаний з жодним із процесів читання;

- сигнали, що посилаються процесу, який виконується в режимі задачі, наприклад сигнал тривоги (alarm), що посилається після закінчення визначеного періоду часу, чи довільні сигнали, якими обмінюються процеси за допомогою функції **kill**;

- сигнали, зв'язані з термінальною взаємодією, наприклад із зависанням термінала (коли сигнал-носій на термінальній лінії припиняється з будь-якої причини) чи з натисканням клавіш BREAK і DELETE на клавіатурі термінала;

- сигнали, за допомогою яких процес виконується в режимі трасування.

Концепція сигналів має кілька аспектів, пов'язаних з тим, яким чином ядро посилає сигнал процесу, як процес обробляє сигнал і керує реакцією на нього. Посилаючи сигнал процесу, ядро встановлює в одиницю розряд у полі сигналу в запису таблиці процесів, що відпові-

дає типу сигналу. Якщо процес знаходиться в стані призупинення з пріоритетом, що допускає переривання, ядро відновить його виконання. На цьому роль відправника сигналу (процесу чи ядра) вичерпується. Процес може запам'ятовувати сигнали різних типів, але не має можливості запам'ятовувати кількість одержуваних сигналів кожного типу. Наприклад, якщо процес одержує сигнал про зависання чи про видалення процесу із системи, він встановлює в одиницю відповідні розряди в полі сигналів таблиці процесів, але не можна сказати, скільки екземплярів сигналу кожного типу він одержав.

Ядро перевіряє одержання сигналу, коли процес збирається перейти з режиму ядра в режим задачі, а також коли він переходить у стан призупинення, чи виходить з цього стану з досить низьким пріоритетом планування. Ядро обробляє сигнали тільки тоді, коли процес повертається з режиму ядра в режим задачі. Таким чином, сигнал не робить негайного впливу на поводження процесу, що виконується в режимі ядра. Якщо процес виконується в режимі задачі, а ядро тим часом обробляє переривання, що було приводом для надсилання процесу сигналу, ядро розпізнає й обробить сигнал, коли вийде з переривання. Таким чином, процес не буде виконуватися в режимі задачі, поки сигнали залишаються неопрацьованими.

4.6. Механізм керування процесами на рівні користувача засобами командної мови

В операційній системі UNIX є дві можливості керування процесами – з використанням командної мови (того чи іншого варіанта shell) і з використанням мови програмування з безпосереднім використанням системних викликів ядра операційної системи. Оскільки процеси є об'єктами, з якими в основному працює операційна система, то кількість утиліт командного інтерпретатора для роботи з процесами мала. У табл. 4.2 подано основні утиліти. Більш докладну інформацію містить електронний довідник man(1).

Таблиця 4.2

Основні утиліти для роботи з процесами

Утиліта	*Призначення утиліти*
Nice –[[-[n] command	Запуск програми на виконання зі статичним пріоритетом **n**, відмінним від прийнятого за замовчуванням; користувач може тільки зменшувати рівень пріоритету, підвищувати його дозволено тільки адміністратору

Закінчення таблиці 4.2

Утиліта	*Призначення утиліти*
renice new_nice pid	Зміна пріоритету процесу під час його виконання; користувач може тільки зменшувати рівень пріоритету, підвищувати його може тільки адміністратор
ps [-опції]	Виведення інформації про існуючі процеси; повнота та вид інформації задаються опціями
Kill [signo] pid1, pid2…	Посилання сигналу **signo** процесам з ідентифікаторами **pid1**, **pid2,**…
at[opt] час_запуску	Зчитування команди стандартного потоку введення і групування їх у завдання **at**, що буде виконано в зазначений час **час_запуску**

4.7. Висновки

Розгляд цієї теми включає:

1) поняття процесів у системі UNIX, які можуть знаходитися в різних логічних станах і переходити зі стану в стан відповідно до встановлених правил переходу, при цьому інформація про стан зберігається в таблиці процесів і в адресному просторі процесу;

2) поняття контексту процесу, що складається з користувальницького контексту (програмний код процесу, дані, стек задачі й області поділюваної пам'яті) і системного контексту, що складається зі статичної частини (запис у таблиці процесів, адресний простір процесу й інформація, що необхідна для відображення адресного простору) і динамічної частини (стек ядра і збережений стан регістрів попереднього контекстового рівня системи);

3) основні принципи керування процесами, деякі утиліти для керування процесами.

Бібліографічний список

1. Бэкон Дж. Операционные системы / Дж. Бэкон, Т. Харрис. – К.: Изд. группа BHV; СПб.: Питер, 2004. – 800 с.

2. Джонсон М.К. Разработка приложений в среде Linux / М.К. Джонсон, Э.В. Троан; пер. с англ. – М.: ООО «И.Д. Вильямс», 2007. – 544 с.

3. Робачевский А.М. Операционная система UNIX / А.М. Робачевский. – СПб.: БХВ-Санкт-Петербург, 1999. – 528 с.

4. Шеховцов В.А. Операційні системи / В.А. Шеховцов. – К.: Вид. група BHV, 2005. – 576 с.

Інформація в INTERNET

1. Bach M.J. The Design of the UNIX Operating System. – Englewood: Cliffs; Nj: Prentice-Hall, 1986. (Пер. с англ.: Бах М. Архитектура операционной системы UNIX). URL: http://www.lib.ru/BACH

2. Unixhelp for Users. http://www.winerweb.com/UNIX

3. Журнал UnixWorld. http://www.unixword.com/unixword/

4. Кузнецов С.Д. Операционная система UNIX. URL: http://www.citforum.ru/operating_system/unix/contents.shtml

5. Официальный сервер UNIX. http://www.rdg.opengroup.org/unix/

Риженко Олена Іванівна
Шевель Володимир Вікторович

ОСНОВИ АРХІТЕКТУРИ ОС UNIX:
РОБОТА В СЕРЕДОВИЩІ КОМАНДНОГО ІНТЕРПРЕТАТОРА bash

Редактор Є.О. Александрова

Зв. план, 2009
Підписано до друку 27.05.2009
Формат 60x84 1/16. Папір офс. №2. Офс. друк
Ум. друк. арк. 4,5. Обл.-вид. арк. 5,12. Наклад 50 прим.
Замовлення 185. Ціна вільна

Національний аерокосмічний університет ім. М.Є.Жуковського
"Харківський авіаційний інститут"
61070, Харків-70, вул. Чкалова,17
http://www/khai/edu
Видавничий центр "ХАІ"
61070, Харків-70, вул. Чкалова,17
izdat@khai.edu