Міністерство освіти і науки України
ДВНЗ "Прикарпатський національний університет імені Василя Стефаника"
Кафедра комп'ютерної інженерії та електроніки

Курс лекцій з дисципліни

"СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ"

Затверджено
на засіданні кафедри радіофізики і електроніки
протокол № 4 від 8 листопада 2017 р

Розроблено:
доц. Голота В.І.

Івано-Франківськ 2017

Зміст

1. ПРИНЦИПИ І МЕТОДОЛОГІЯ СТВОРЕННЯ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 3
2. ОПЕРАЦІЙНІ СИСТЕМИ 21
3. СИСТЕМИ ДИНАМІЧНОГО ТА СТАТИЧНОГО ПЛАНУВАННЯ 57
4. ОРГАНІЗАЦІЯ ОПЕРАТИВНОЇ ПАМ’ЯТІ 68
5. ОРГАНІЗАЦІЯ ФАЙЛОВИХ СИСТЕМ 78
6. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РОЗРОБЛЕННЯ СПЗ 108
7. КОМАНДИ ОС LINUX 122
8. МОВА СЦЕНАРІЇВ BASH 143
9. СТВОРЕННЯ І ВИКОРИСТАННЯ ФУНКЦІЙ У BASH 177
10. ПЕРЕНАПРАВЛЕННЯ ВВЕДЕННЯ-ВИВЕДЕННЯ У BASH 186
11. ПРОЦЕСИ І СИГНАЛИ 191
12. ТЕКСТОВІ СПИСКИ ВИБОРУ У BASH. КОМАНДА DIALOG 203
13. ПОТОКОВІ РЕДАКТОРИ SED І GAWK 213
СПИСОК ЛІТЕРАТУРИ 223

1. ПРИНЦИПИ І МЕТОДОЛОГІЯ СТВОРЕННЯ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Мета. Ознайомити студентів із принципами побудови і базовими поняттями системного програмного забезпечення.

Вступ. Системне програмне забезпечення будується на основі тих же принципів, на яких будується і прикладне програмне забезпечення. Системне програмне забезпечення – це комплекс програмних засобів, які забезпечують керування як компонентами комп'ютерної системи, так і задачами користувачів. Предметом дисципліни системне програмне забезпечення є теоретичні і практичні питання пов'язані з проектуванням, розробленням та експлуатацією програмного забезпечення комп'ютерних систем на різних апаратних платформах. Архітектура системного програмного забезпечення (software architecture) відображає набір структурних і функціональних елементів, а також відношень між ними.

План.
1. Принципи побудови системного програмного забезпечення
2. Базові поняття і класифікація програмного забезпечення
3. Предмет та задачі системного програмного забезпечення
4. Засоби розроблення програм
5. Інтерфейси і стандарти
5.1. Інтерфейс API
5.2. Інтерфейс ABI
5.3. Стандарти
6. Розроблення програмного забезпечення
7. Авторські права і ліцензії на програмне забезпечення

1. Принципи побудови системного ПЗ

Принципи побудови системного програмного забезпечення (ПЗ) такі ж, як і принципи побудови прикладного програмного забезпечення. Серед них можна виділити наступні:

Частотний принцип – базується на розділенні програм і даних по частоті використання. Для операцій які часто використовуються створюються умови їх швидкого виконання. Найбільш часто виконувані операції створюються найкоротшими. До найбільш часто виконуваних даних забезпечується найшвидший доступ.

Принцип модульності. *Модуль* – це функціональний елемент комп'ютерної системи, який має певне оформлення, наповнення та закінчення в рамках даної системи, а також засоби взаємозв'язку з модулями різних рівнів своєї або іншої системи. За своїм визначенням модуль вказує на легкий спосіб його заміни іншим при наявності певних програмних інтерфейсів. Системні програми розділяються на модулі за функціональним призначенням. Як правило, модулі впорядковані ієрархічно, що дозволяє значно спростити експлуатацію та розробку програм.

Модулі можуть бути наступними:

• Привілейований модуль працює без активації системи переривань. Модуль виконується до кінця, після чого його можна знову викликати для виконання іншого завдання. Привілейований модуль, який діє як позмінно спільно використовуваний ресурс, виконується у такій послідовності: виключення переривань – виконання модуля – включення переривань. Прикладом таких модулів є модулі ядра ОС.

• Непривілейований модуль може бути перерваний під час роботи, а результати, отримані перед перериванням, можуть втратитися.

• Одноразово використовуваний модуль запускається на виконання один раз. Для повторного виконання його потрібно повторно запустити.

• Реентерабельні модулі дозволяють багаторазове переривання і повторний запуск з інших процесів. Вони забезпечують зберігання проміжних результатів, отриманих до переривання, і повернення до них, коли обчислювальний процес відновлюється з перерваної точки. Така можливість реалізується за допомогою статичних або динамічних методів виділення пам'яті під проміжні дані.

• Модулі з повторним входженням допускають багаторазове паралельне виконання, але без переривань.

Принцип функціональної важливості. Частина важливих модулів повинна бути постійно в активному стані з метою ефективної реалізації обчислювального процесу. Така частина модулів системної програми називається *ядром*. При формуванні ядра необхідно розв'язати дві протилежності: з однієї сторони в склад ядра мають входити програми які використовуються найчастіше, з іншої сторони – розмір ядра повинен бути мінімальний.

Принцип генерованості. Системні програми мають налаштовуватися з врахуванням конкретної апаратної конфігурації обчислювальної системи та кола розв'язуваних задач. Цей принцип реалізується окремою програмою чи утилітою, в результаті чого генерується потрібна версія системної програми.

Принцип функціональної надлишковості. Системна програма повинна мати можливість виконувати одну і ту ж саму функцію різними способами.

Принцип за замовчуванням. У складі системної програми мають зберігатися певні базові описи модулів, конфігурацій та даних, які визначають прогнозовані параметри апаратного та програмного забезпечення.

Принцип переміщуваності. Модулі системної програми мають проектуватися і розроблятися так, щоб їх виконання не залежало від розміщення в оперативній пам'яті. Модуль, на конкретне розміщення в оперативній пам'яті, налаштовується перед виконанням програми і при цьому визначаються фактичні адреси команд програми в залежності від типу обчислювальної системи та моделі пам'яті.

Принцип захисту. Необхідно розробляти засоби, які захищають програми і дані користувача від ушкоджень та будь-якого впливу інших користувачів або програм. Програми повинні бути захищені як на етапі виконання, так і під час зберігання. Цей принцип в тій чи іншій мірі реалізований в кожній багатозадачній ОС.

Принцип незалежності програм від зовнішніх пристроїв. Дозволяє здійснювати управління та обмін даними із зовнішніми пристроями незалежно від їх конкретних фізичних характеристик. Цей принцип в багатьох сучасних системах реалізований за допомогою механізму драйверів.

*Принцип відкритості і нарощуваності***.** Відкрита системна програма має бути відкритою і доступною для аналізу спеціаліста. Нарощувана програма дозволяє виконувати не лише принцип генерованості, але й дозволяє вводити в склад системи нові модулі і нарощувати існуючі.

Принцип сумісності. Це здатність ПЗ виконуватися на інших апаратних платформах. Двійкова сумісність досягається при запуску виконуваної програми на іншій ОС. Для цього потрібна сумісність на рівні команд процесора, системних викликів та на рівні викликів динамічно зв'язуваних бібліотек DLL. Двійкова сумісність на рівні різних процесорів потребує емуляції бібліотечних функцій та окремих команд за допомогою підпрограм.

Сумісність на рівні сирцевих текстів потребує наявності відповідного транслятора у складі ПЗ і сумісності на рівні бібліотек та системних викликів. При цьому необхідно перекомпілювати сирцеві коди програм у новий виконуваний модуль.

Дотримання принципу сумісності можливе при дотриманні якогось стандарту, наприклад POSIX. У ньому стандартизовані звернення до API, файлова система, організація доступу до зовнішніх пристроїв, набір системних команд (моніторів).

2. Базові поняття і класифікація програмного забезпечення

В загальному випадку обчислювальна система складається з двох взаємодоповнюючих і взаємодіючих елементів: апаратного і програмного забезпечення.

В склад апаратного забезпечення входять: *мікропроцесор* (який складається з арифметико-логічного пристрою ((АЛП) Arithmetic-logic unit, ALU), регістрів та пристрою керування), *оперативний запам'ятовуючий пристрій, периферійні пристрої.*

Оперативний запам'ятовуючий пристрій складається з комірок пам'яті, кожна з яких має свою адресу. Комірки пам'яті містять інформацію, яка може інтерпретуватися мікропроцесором як команди або дані. Периферійні або зовнішні пристрої здійснюють введення/виведення та зберігання інформації.

Програмне забезпечення (ПЗ/Software) – комп'ютерні програми, процедури, а також документація й дані, що з ними асоційовані, які стосуються функціонування комп'ютерної системи.

Програма – дані, призначені для управління конкретними компонентами системи обробки інформації з метою реалізації певного алгоритму, послідовність машинних команд, призначена для досягнення конкретного результату.

Програмування (Programming) – процес підготовки задач для їх розв'язання за допомогою комп'ютера; ітераційний процес складання програм.

ПЗ є одним із видів забезпечення обчислювальної системи, поряд з технічним (апаратним), математичним, інформаційним, організаційним і методичним забезпеченням.

ПЗ за призначенням поділяється на:

- прикладне;
- системне;
- інструментальне.

Прикладне ПЗ (application, application software) – це сукупність програм призначених для вирішення конкретних задач фахової діяльності користувача.

Системне ПЗ (system software) – призначене для управління роботою комп'ютера, розподілу його ресурсів, підтримки діалогу з користувачами, а також для часткової автоматизації розроблення нових програм. Як правило, системні програми забезпечують взаємодію інших програм з апаратними складовими, організацію інтерфейсу користувача.

Виділяють три типи системного ПЗ:

• *операційна система* (ОС) – програмне забезпечення, яке забезпечує інфраструктуру, на якій можуть працювати прикладні програми. Найпоширеніші ОС – Linux, Microsoft Windows, Mac OS X;

• *системи програмування* – призначені для часткової автоматизації процесу розроблення та налагодження програм;

• *службові програми* (утиліти) розширюють можливості ОС.

Найбільш поширені службові програми:

1. Диспетчери файлів (файлові менеджери). За їх допомогою виконується більшість операцій з обслуговування файлової структури: копіювання, переміщення, перейменування файлів, створення каталогів, знищення об'єктів, пошук файлів та навігація у файловій структурі. Ці базові програмні засоби містяться у складі програм системного рівня і встановлюються разом з ОС.

2. Засоби стиснення даних (архіватори). Призначені для створення архівів. Архівні файли мають підвищену щільність запису інформації і відповідно ефективніші для зберігання та перенесення інформації.

3. Засоби діагностики. Призначені для автоматизації процесів діагностування програмного та апаратного забезпечення. Їх використовують для виправлення помилок і оптимізації роботи комп'ютерної системи.

4. Програми інсталяції (установлення). Призначені для контролю за додаванням у поточну програмну конфігурацію нового ПЗ. Вони слідкують за станом і зміною програмного

середовища, відслідковують та протоколюють утворення нових зв'язків, загублені під час знищення певних програм. Прості засоби керування встановленням та знищенням програм містяться у складі ОС, але можуть використовуватись і додаткові службові програми.

5. Засоби комунікації. Дозволяють установлювати з'єднання з віддаленими комп'ютерами, передають повідомлення електронної пошти, пересилають факсимільні повідомлення тощо.

6. Засоби перегляду та відтворення. Застосовують переважно для роботи з файлами, їх завантажують у "рідну" прикладну систему і вносять необхідні зміни.

7. Засоби комп'ютерної безпеки. До них належать засоби пасивного та активного захисту даних від пошкодження, несанкціонованого доступу, перегляду та зміни даних. Засоби пасивного захисту – це службові програми, призначені для резервного копіювання. Засоби активного захисту застосовують антивірусне ПЗ. Для захисту даних від несанкціонованого доступу, їх перегляду та зміни використовують спеціальні системи, базовані на криптографії.

Інструментальне ПЗ – програми, призначені для розроблення всіх видів інформаційно-програмного забезпечення. До інструментального ПЗ відносяться:

- транслятори мов програмування;
- інструменти розробника (software development kit);
- системи контролю версій;
- системи відслідковування помилок;
- текстові редактори;
- системи керування базами даних.

Під архітектурою ПЗ (software architecture) розуміють структуру програмних компонент, а також відношення між ними.

3. Предмет та задачі системного програмного забезпечення

Предметом (метою) дисципліни системного програмного забезпечення є теорія і практика проектування, розроблення і функціонування системного ПЗ (операційна система і системи програмування, а також їх елементи), яке забезпечує роботу обчислювальної системи.

Задачею системного ПЗ є забезпечення виконання задач користувачів при ефективному використанні апаратних ресурсів обчислювальної системи.

Системне ПЗ виступає як "міжшаровий інтерфейс" між апаратурою і застосуваннями користувача. Системне ПЗ не розв'язує конкретні практичні задачі, а тільки забезпечує роботу інших програм, надаючи їм сервісні функції. Системне ПЗ включає в себе дві компоненти:

- операційну систему;
- системні програми.

Необхідно розрізняти поняття операційної системи і операційного середовища.

Операційна система (ОС, Operation System) виконує функції керування обчислювальними процесами в комп'ютерній системі та розподіляє ресурси між цими процесами. ОС є комплексом керуючих і обробляючих програм які забезпечують технічне функціонування обчислювальної системи, діагностику несправностей, планування використання ресурсів системи та виконання задач користувачів, сформульованих у термінах обчислювальної машини. ОС забезпечує введення/виведення інформації та обмін даними між різними компонентами системи. Як правило, ОС розглядають як продовження апаратної частини комп'ютера. Тому ще однією задачею ОС є керування виконанням завдань користувачів з метою максимального підвищення продуктивності обчислювальної системи.

В логічній структурі обчислювальної системи ОС займає проміжне положення між пристроями з їх мікроархітектурою, машинними мовами, мікропрограмами з однієї сторони, та прикладними програмами з іншої сторони. Місце системного ПЗ в складі КС показано на рис.1.

Прикладні програми
Додаткове системне ПЗ
Програмне середовище користувача
Ядро ОС
Вбудоване ПЗ (firmware)
Системне ПЗ
ОС
Програмне забезпечення

Архітектура набору команд
Мікропрограма
Мікроархітектура
Цифровий логічний рівень
Фізичні пристрої
Апаратна платформа (архітектура комп'ютера)

Рисунок 1 – Місце системного ПЗ в структурі КС

Операційне середовище – це набір сервісів і правил звернення до них, інтерфейси необхідні для взаємодії з ОС. Операційне середовище створюють системні програми. До *системних програм* відносяться:

– вбудовувані програми;
– системи керування файлами;
– утиліти:
  – для роботи з реєстрами;
  – для моніторингу обладнання;
  – для тестування обладнання;
  – дискові (для дефрагментації, очистки і розмітки диску, резервного копіювання і стискання дисків).
– інтерфейсні оболонки;
– системи програмування;
– диспетчери реального часу;
– драйвери (програми керування пристроями введення-виведення).

Вбудовані програми (firmware) – це програми, записані у постійну пам’ять цифрових електронних пристроїв. У деяких випадках, вбудовані програми (як BIOS IBM PC комп’ютерів), є по суті частиною ОС, яка зберігається у постійній пам’яті. У достатньо простих пристроях вся ОС може бути вбудованою.

Системи керування файлами призначені для організації зручного доступу до даних, структурованих певним чином і дозволяють замінити низькорівневий доступ з фізичною адресацією даних на високорівневий з логічною адресацією.

Утиліта (utility, tool) – це спеціальне системне ПЗ, яке виконує ряд сервісних функцій, як по обслуговуванню самої ОС, так і по підготовці носіїв, оптимізації розміщення даних.

Інтерфейсні оболонки розширюють можливості взаємодії з ОС. До цього класу ПЗ відносяться засоби організації іншої ОС в рамках віртуальної машини засобами даної ОС, а також емулятори ОС, коли одна ОС може бути запущена в рамках іншої ОС.

Системи програмування (development system) – призначені для автоматизації процесу розроблення, супроводження та налагодження програм.

4. Засоби розроблення програм

До засобів розроблення програм відносяться:

– асемблери, програми, які перетворюють програми на мові асемблера у машинні коди у формі об’єктного модуля;

– транслятори, програми, які перетворюють текст на мові високого рівня в еквівалентну програму на машинній мові;

– інтерпретатори, програми, які аналізують команди або інструкції програми і тут же виконують їх;

– компонувачі (редактори зв'язків, linker), програми, які виконують компонування: приймають на вхід один або декілька об'єктних модулів і збирають з них виконуваний модуль;

– завантажувачі (loader), програми які звантажують програму в ОП для її виконання;

– налагоджувачі, наладники (debugger) – модулі середовища розробки або окремі програми, призначені для пошуку помилок у програмах;

– текстові редактори, програми, які дозволяють створювати і редагувати початковий код програми (текстовий редактор може бути окремим застосуванням або бути вбудованим в інтегроване середовище розробки);

– бібліотеки підпрограм, це колекції програм або об'єктів, які використовуються при розробленні ПЗ.

В комп'ютерних системах (КС) можуть використовуватися процесори з різними системами *машинних команд*. КС виконують програми, які переведені у машинні команди. Так як машинні команд є процесорно орієнтовані, то вони не використовуються у програмуванні. При розробленні системного ПЗ використовуються різні типи мов програмування:

- мови низького рівня – асемблер, макроасемблер (Assembler, Macro Assembler);
- мови сценаріїв (Perl, Bash, Python, Ruby);
- мови високого рівня (C/C++, C#, Java, ADA);

У мові *асемблера* замінено цифрові коди машинних команд на відповідні мнемонічні (букво або букво-цифрові) позначення і використовуються символьні імена даних. При переведенні на машинні команди кожна інструкція, яка визначає відповідну машинну команду замінюється цифровим кодом цієї інструкції.

Мова *макроасемблер* поряд з мнемонічними позначеннями множин команд (наприклад, `mul`, `add`) допускає використання спеціальних макрокоманд, які не мають прямих аналогів в машинних командах. При перекладі на машинні команди, інструкція, яка позначає макрокоманду, замінюється групою машинних команд.

Мови *сценаріїв* дозволяють записувати і виконувати послідовності команд ОС і інструкцій сценаріїв, аналізувати коди їх завершень, організовувати галуження і цикли в роботі сценарію.

Мови *високого рівня* дозволяють описувати більшість алгоритмів у зручній формі, подібно до звичайного запису математичних дій, що зменшує трудомісткість програмування.

Мови програмування обробляють сирцеві програм. *Сирцева програма* (початкова програма, source code) – написана на одній з мов програмування і складається з послідовних інструкцій. Сирцева програма (або декілька програм) записаних на зберігання у файл називається *сирцевим модулем*.

Сирцева програма, написана на мові програмування високого рівня складається з *конструкцій* (statement), *виразів* (expression), *операцій* (operator) і *операндів* (operand).

Операнди – елементи даних над якими виконується операція.

Операція позначає вбудовані в мову класи операцій (арифметичні, логічні, порозрядні, порівняння та інші), які виконуються над даними.

Вираз складається з атомарних елементів даних (безпосередні дані, константи, змінні, функції) та операцій, який закінчується “;” і обчислює нове значення.

Конструкція (інструкція) – найменша автономна частина програми, яку можна виконати окремо. Конструкції можуть бути прості або складні і містити вирази. Конструкції поділяють на наступні категорії: умовні, циклу, інструкції переходів, позначки, інструкції-вирази, блоки.

Сирцева програма має бути перетворена у форму придатну для машинної обробки. Таке перетворення називається *трансляцією*. Загальна схема трансляції показана на рис. 1.2.

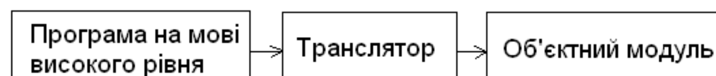

Рисунок 2 – Загальна схема трансляції

Програма, яка перетворює сирцевий модуль в об’єктний на мові низького рівня або машинній мові називається *транслятором*.

В залежності від вхідної мови та порядку трансляції і виконання інструкцій програми всі транслятори поділяються на асемблери, інтерпретатори, компілятори.

Асемблер – це транслятор з мови низького рівня. Сирцевий модуль перетворюється асемблером у об’єктний модуль, який є особливим записом об’єктної програми. *Об’єктний модуль* містить машинні команди та інформацію необхідну для об’єднання цього модуля з іншими незалежно-трансльованими модулями, а також інформацію необхідну для розміщення цього модуля в оперативній пам’яті. Безпосередньо цей модуль не є виконуваною програмою – тому він потребує додаткової обробки компонувачем.

Транслятор з мови високого рівня називається інтерпретатором або компілятором в залежності від порядку здійснення етапів трансляції і виконання інструкцій програми. Схеми інтерпретації та компіляції програм показані на рис. 3 і 4.

Інтерпретатор після трансляції кожної окремої інструкції забезпечує її виконання. Тобто етапи трансляції і виконання почергово повторюються.

Компілятор транслює всі інструкції програми, а виконання програми в цілому відбувається без його участі. Тобто етапи трансляції окремих інструкцій здійснюються безпосередньо одна за одною і вони повністю ізольовані від процесу виконання програми.

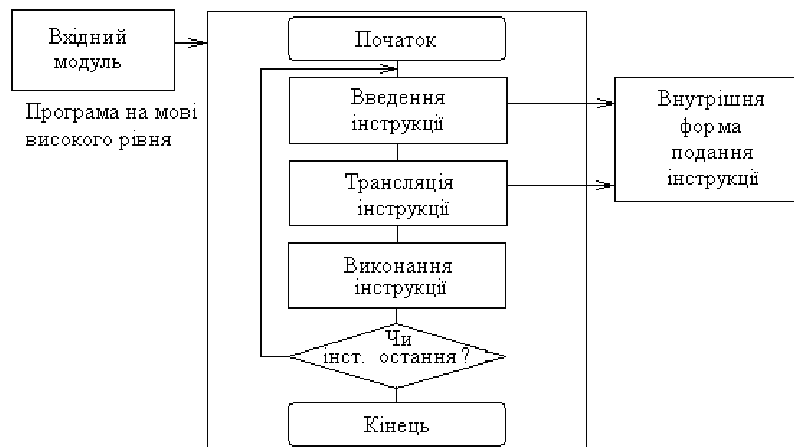

Рисунок 3 – Схема інтерпретації програми

Рисунок 4 – Схема компіляції програми

Під час формування програми компілятор підключає об’єктні модулі, які обчислюють значення математичних функцій, виконують операції введення/виведення, тощо. Ці модулі зберігаються в бібліотеках підпрограм компілятора і автоматично вибираються з них компонувачем.

Компонувач об’єднує декілька об’єктних модулів в один виконуваний (завантажувальний, бінарний) модуль, який готовий до безпосереднього виконання після завантаження і розміщення в оперативній пам’яті, рис. 1.5.

Необхідність об’єднання об’єктних модулів зумовлена наявністю майже в кожному модулі звернень до інших об’єктних модулів, які транслюються незалежно один від одного і зберігаються в бібліотеці підпрограми.

Компонувач назначає кожній машинній команді і кожному елементу даних певне місце в оперативній пам’яті і забезпечує об’єктним модулям можливість звернень між собою.

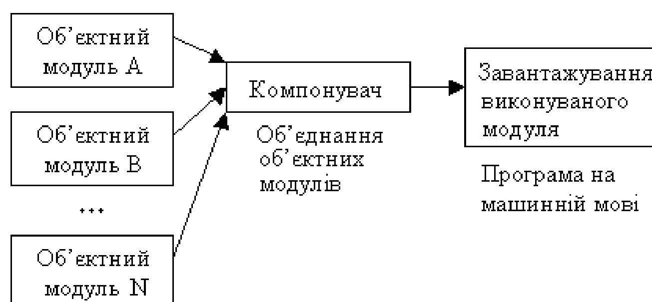

Рисунок 5 – Схема компонування виконуваного модуля

Для спрощення і пришвидшення пошуку і виправлення помилок в програмі користувача використовуються налагоджувачі програм. *Налагоджувач* орієнтований на сумісну роботу з програмою, яка написана на певній мові програмування. Налагоджувач має засоби для перегляду та зміни значень змінних програми в оперативній пам’яті, оперативно керує ходом виконання програми, реалізує інші шляхи пошуку помилок в діалоговому режимі.

5. Інтерфейси і стандарти

Системне ПЗ має бути *переносимим*, тобто виконуватися на різних апаратних платформах і різних версіях ОС, як старих, так і нових. Для забезпечення переносимості ПЗ на системному рівні є два окремих набори визначень і описань. Перший з них – це інтерфейс прикладного програмування, інтерфейс програмування застосунків (Application Programming Interface, **API**), а другий – двійковий інтерфейс застосунків (Application Binary Interface, **ABI**). Обидва визначають і описують інтерфейси між різними частинами програмного забезпечення.

5.1. Інтерфейси API

Інтерфейс *API* – є набором готових класів, процедур, функцій і констант, які надаються застосунком або операційною системою для використання у зовнішніх прикладних програмах. Відповідно, API може реалізовуватися на рівні ОС (являючи собою системні виклики) або на рівні систем програмування (як бібліотеки RTL). Крім того, API може реалізовуватися як зовнішні бібліотеки, наприклад MFC, VCL.

API визначає функціональність, яку надає програма, модуль, бібліотека, абстрагуючи від того, як ця функціональність реалізована.

API бібліотеки функцій і класів включає в себе описання сигнатур і семантику функцій. Сигнатура функції – це частина загального оголошення функції, яка дозволяє ідентифікувати її серед інших функцій. Семантика функції – це описання того, що дана функція робить.

В інтерфейсі API визначаються способи, за допомогою яких один фрагмент ПЗ взаємодіє з іншим на рівні сирцевих кодів. Він надає абстракцію у вигляді стандартних інтерфейсів (звичайно функцій), які одна частина ПЗ (високорівнева) може викликати з іншої частини (низькорівневої). В API просто визначається *інтерфейс*. Фрагмент ПЗ, який фактично надає інтерфейс API, називається *реалізацією* API.

API гарантує, що якщо обидві частини ПЗ будуть задовольняти API, то вони будуть *сумісними на рівні сирцевого коду*. Це означає, що застосунки користувача будуть успішно компілюватися з даною реалізацією API.

Якщо API різних платформ співпадає, то код для цих платформ можна компілювати без змін.

Реальним прикладом є API, який визначений у стандарті мови Сі і реалізований у стандартній бібліотеці Сі. Він визначає родини базових і обов'язкових функцій.

Платформо незалежний системний інтерфейс для комп'ютерних середовищ описується стандартом POSIX (Portable Operating System Interface For Computer Environment). Стандарт визначає мінімальний набір системних викликів (більше 100) для відкритих ОС, що базуються на UNIX системах.

На відміну від UNIX, у Windows системні виклики і для їх виконання бібліотечні виклики повністю розділені. Для виклику служб ОС використовується набір процедур Win32 API (декілька тисяч), багато з яких працюють в просторі користувача. В Unix графічний інтерфейс користувача запускається в просторі користувача, а у Windows – в режимі ядра.

Таблиця 1.1 – Основні системні виклики

POSIX	Призначення	Win32 API
fork	створити дочірній процес, ідентичний батьківському	CreateProcess(fork+execve)
waitpid	очікувати завершення дочірнього процесу	WaitForSingleObject
execve	перемістити образ пам'яті процесора	-
exit	завершити виконання процесу	ExitProcess
open	відкрити файл	CreateFile
close	закрити файл	CloseHandle
read	читання із файлу у буфер	ReadFile
write	записування даних з буфера у файл	WriteFile
lseek	перемістити вказівник файлу	SetFilePointer
stat	інформація про стан файлу	GetFileAttributesEx
mkdir	створити каталог	CreateDirectory
rmdir	вилучити каталог	RemoveDirectory
link	створити новий елемент каталогу, який посилається на інший	-
unlink	вилучити елемент каталогу	DeleteFile
mount	монтування файлової системи	-
umount	демонтування файлової системи	-
chdir	змінити робочий каталог	SetCurrentDirectory
chmod	змінити біти захисту файлу	-
kill	послати сигнал процесу	-
time	отримати системний час	GetLocalTime

5.2. Інтерфейс ABI

Інтерфейс *ABI* – є набором домовленостей для доступу застосунків до ОС та інших низькорівневих сервісів, спроектований для переносимості виконуваного коду між платформами, які мають сумісні API. На відміну від API, який регламентує сумісність на рівні сирцевого коду, ABI можна розглядати як набір правил, який дозволяє компонувачу об'єднувати відкомпільовані модулі компоненти без перекомпіляції усього коду.

Двійковий інтерфейс за стосунків регламентує:

- використання регістрів процесора;
- склад і формат системних викликів та викликів одного модуля іншим;
- формат передачі аргументів і значення, яке повертається при виклику функції.

Двійковий інтерфейс застосунків описує функціональність, яку надає ядро ОС і архітектура набору команд (ISA – instruction set architecture).

Любий ABI прив'язаний до конкретної машинної архітектури і реалізується ланцюжком інструментів – компілятор, компонувач, завантажувач. Якщо API і AVI різних платформ співпадають, то виконувані файли можна переносити на ці платформи без змін.

Місце інтерфейсів в логічній структурі КС:

```
Застосунки
   API
Бібліотеки
   AVI
ОС
   ISA
Апаратура
```

5.3. Стандарти

Стандарти створюються для впорядкування процесу розроблення ПЗ. Комерційні організації розробляють стандарти для свого внутрішнього використання. Державні організації і промислові консорціуми підтримують ініціативи з розроблення стандартизованих офіційних стандартів. Так IEEE підтримує POSIX – набір стандартів, які описують інтерфейси між ОС та ПЗ. Стандарт створений для забезпечення сумісності різних Unix-подібних ОС та переносимості прикладних програм на рівні сирцевого коду. Формально стандарт визначений як IEEE 1003, назва міжнародного стандарту ISO/IEC 9945.

Стандарт складається з чотирьох основних розділів:

• Основні визначення – список основних визначень і угод, які використовуються в специфікаціях, список заголовкових файлів мови Сі, які мають бути надані відповідно стандарту системою.

• Оболонка і утиліти – опис утиліт і командної оболонки shell, стандарти регулярних виразів.

• Системні інтерфейси – список системних викликів мови C.

• Обґрунтування – пояснення принципів, які використані у стандарті.

В процесі розвитку POSIX створено ряд версій.

1. POSIX.1, Core Services (створення і керування процесами):

• Сигнали.
  - Винятки плаваючої крапки.
  - Порушення сегментації.
  - Неправильна інструкція.
  - Помилка шини.
  - Таймери.
  - Операції з файлами і директоріями.
  - Конвеєри (Pipes).
  - Бібліотека C (стандарт C).
  - Інтерфейс і керування портами введення-виведення.

2. POSIX.1b. Real time extensions (розширення реального часу)
  - Планування пріоритетів.
  - Сигнали реального часу.
  - Годинники і таймери.
  - Семафори.
  - Передача повідомлень.
  - Спільно використовувана пам'ять.
  - Асинхронне та синхронне введення-виведення.
  - Інтерфейс блокування пам'яті.

3. POSIX.1c. Threads extensions (розширення потоків виконання).

• Створення, управління і очистка потоків.
• Планування потоків
• Синхронізація потоків
• Керування сигналами

Виробники ОС Unix утворили консорціум Open Group і розробили специфікацію **SUS** (Single Unix Specification).

Стандартизуються не тільки системні інтерфейси ОС, але також і мови програмування системного рівня. Так компілятор С `gcc` підтримує стандарт ISO 99.

6. Розроблення програмного забезпечення

Життєвий цикл (ЖЦ) програмного забезпечення – це увесь період його розроблення і експлуатації, починаючи з моменту виникнення і закінчуючи припиненням усіх видів її використання. Життєвий цикл включає наступні етапи:

- планування (planning);
- проектування (design);
- реалізація (implementation):
  - програмування, кодування (coding);
  - модульне тестування (unit test);
  - тестування підсистем (subsystem test);
  - інтегруюче тестування системи (integration test);
  - оціночне випробування (evaluation).

Найбільш відомі наступні моделі ЖЦ – класична послідовнна, каскадна, ітеративна, спіральна, формалізована.

Класична послідовна (stagewise model) передбачає послідовне виконання етапів проекту;

Каскадна (*водоспадна*) модель (waterflow model) передбачає послідовне виконання всіх етапів проекту в строго фіксованому порядку. Перехід на наступний етап означає повне завершення робіт на попередньому етапі і неможливість повернення до попередніх етапів, рис. 6.

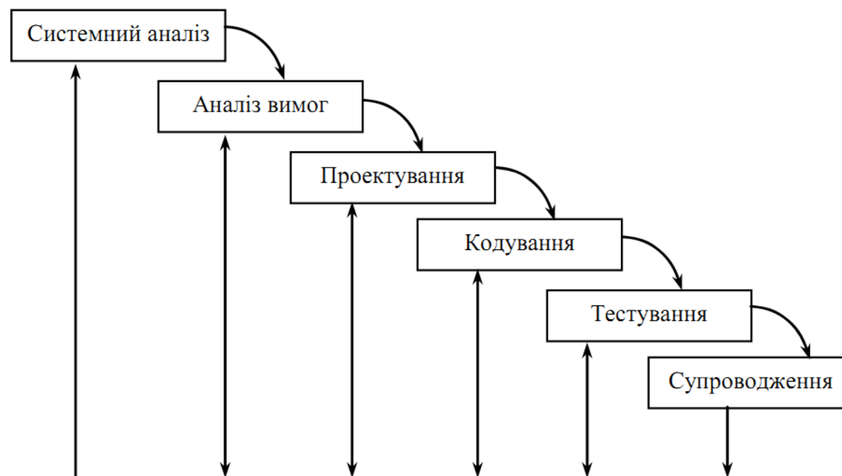

Рисунок 6 – Каскадна модель розроблення ПЗ

Ітеративна модель (iterative and incremental development) передбачає розбиття життєвого циклу проекту на послідовність ітерацій, кожна з яких нагадує “міні-проект” з усіма фазами ЖЦ.

Спіральна модель (spiral model) розроблення ПЗ має вигляд серії послідовних ітерацій. На кожному витку спіралі створюється чергова версія продукту, уточнюються вимоги проекту, визначається його якість і плануються роботи наступного витка. Особлива увага приділяється

початковим етапам розроблення – аналізу і проектуванню, де реалізованість тих чи інших технічних рішень перевіряється і обґрунтовується за допомогою створення прототипів (макетування). Завдяки ітеративній природі спіральна модель допускає коригування у ході роботи, що сприяє поліпшенню продукту. Спіральна модель розроблення ПЗ вимагає визначення ключових контрольних точок проекту (milestones), рис. 7.

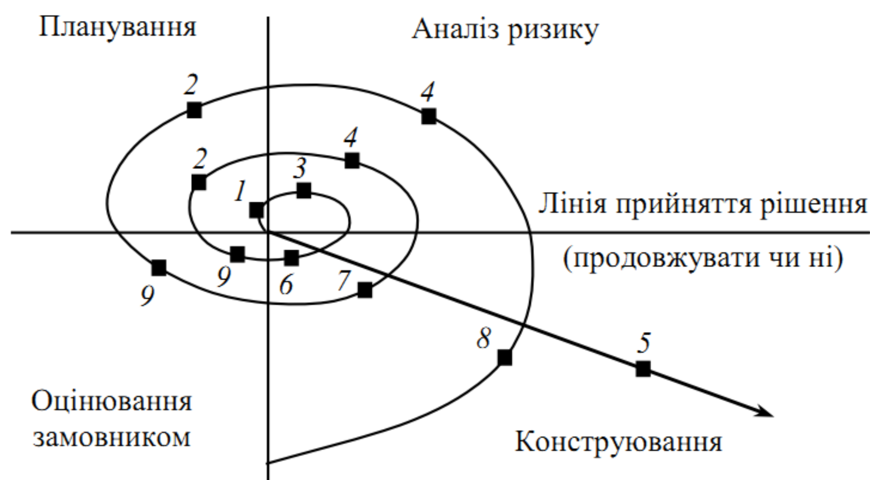

Рисунок 7 – Спіральна модель: 1 – початковий збір вимог та планування проекту; 2 – збір вимог вже на основі рекомендацій замовника; 3 – аналіз ризику на основі початкових вимог; 4 – аналіз ризику на основі реакції замовника; 5 – перехід до комплексної системи; 6 – початковий макет системи; 7 – наступний рівень макета; 8 – побудована система; 9 – оцінювання замовником

Формалізована модель використовує для розроблення ПЗ системи на основі мови UML (наприклад Rational Unified Process (RUP) фірми IBM).

Моделі життєвого циклу розроблення ПЗ дають відповідь на запитання «Що роботи на наступному кроці?", але не кажуть "Як це зробити".

Методологія програмування – сукупність методів, застосовуваних на різних стадіях життєвого циклу програмного забезпечення, що мають спільний філософський підхід та, відповідно до цього підходу, дозволяють забезпечити найкращу ефективність.

Кожна методологія характеризується:

- філософським підходом або основними принципами. Ці принципи, від яких залежить ефективність всієї методології, звичайно можна коротко сформулювати і легко пояснити;
- узгодженою множиною моделей та методів, які реалізують дану методологію;
- концепціями (поняттями), що дозволяють більш точно визначити методи.

Найбільше поширення отримали наступні методології розроблення ПЗ:

Гнучке розроблення ПЗ (Agile) – клас методологій розроблення ПЗ на основі ітеративної моделі життєвого циклу, в якій вимоги та рішення еволюціонують через співпрацю між самоорганізовуваними командами.

Сварливе розроблення ПЗ (scrum) – методологія, що використовує короткі цикли в 2-4 тижні для підготовки повністю відтестованих і готових для оцінки замовником версій програм;

Екстремальне розроблення ПЗ (eXtreme Programming, XP) – методологія, що приділяє основну увагу ефективній комунікації між замовником і виконавцем упродовж усього проекту з розроблення ПЗ для відслідковування зміни вимог. В основу підходу покладена командна робота та постійне тестування прототипів.

Термінове переороблення ПЗ (software triage) – кардинальне перероблення проекту під керівництвом ветеранів.

Раціонально уніфіковане розроблення ПЗ на базі UML (Rational Unified Process (RUP) – пропонує ітеративну модель розроблення, що включає чотири фази: початок, дослідження,

побудову та впровадження. Проходження через чотири основні фази називається циклом розроблення, кожен цикл завершується генерацією працездатної версії системи. У ході супроводу продукт продовжує розвиватися і знову проходить ті самі фази. Розроблення ПЗ на базі RUP базується на створенні та супроводі моделей на базі UML.

Використанням графічного середовища Мікрософт для розроблення ПЗ (Microsoft Solution Framework, MSF) – методологія застосовується розробниками, які використовують продукти Microsoft. Методологія, подібна до RUP, також включає чотири фази: аналіз, проектування, розроблення, стабілізацію, є ітераційною, припускає використання об’єктно-орієнтованого моделювання. MSF порівняно з RUP здебільшого орієнтована на розроблення бізнес-додатків.

Програмування ПЗ. В більш вузькому значенні, коли методологія застосовується на стадії програмування, її зазвичай називають *парадигмою* програмування.

Парадигма програмування – це система дій та понять, які визначають стиль написання програм, а також спосіб мислення програміста.

Серед парадигм програмування виділяють імперативну та декларативну.

Імперативна парадигма – парадигма програмування, згідно з якою описується процес отримання результатів як послідовність інструкцій зміни стану програми. Подібно до того, як з допомогою наказового способу в мовознавстві перераховується послідовність дій, що необхідно виконати, імперативні програми є послідовністю операцій комп’ютеру для виконання. Поширеним синонімом імперативного програмуванню є процедурне програмування.

Декларативна парадигма – парадигма програмування, відповідно до якої, програма описує, який результат необхідно отримати, замість описання послідовності отримання цього результату. Наприклад, веб-сторінки HTML – декларативні, оскільки вони описують, що містить сторінка та що має відображатись – заголовок, шрифт, текст, зображення, але не містить інструкцій як її слід відображати.

Декларативні мови програмування відрізняються від імперативних мов програмування, які вимагають детального описання алгоритму отримання результатів.

Для отримання результатів імперативні програми явно конкретизують алгоритм, а декларативні – явно конкретизують мету і залишають реалізацію алгоритму допоміжному програмному забезпеченню (наприклад, інструкція вибірки SQL конкретизує властивості даних, які слід отримати від бази даних, але не процес отримання цих даних). Програма вважається «декларативною», якщо вона написана винятково функційною або логічною мовою програмування, або мовою обмежень.

Найбільш відомі наступні парадигми програмування процедурна, функційна, структурна, об’єктно-орієнтована, компонентна, логічна.

Процедурне програмування – парадигма програмування, заснована на концепції виклику процедури. Процедури, також відомі як підпрограми, методи або функції. Процедури містять певну послідовність кроків для виконання. В ході виконання програми будь-яка процедура може бути викликана з будь-якого місця програми, включно з самої процедури, яка викликається (рекурсивний виклик).

Функційне програмування – парадигма програмування, яка розглядає програму як обчислення математичних функцій та уникає станів та змінних даних. Функційне програмування основується на застосуванні функцій, на відміну від імперативного програмування, яке основується на змінах в стані та виконанні послідовностей команд.

Іншими словами, функційне програмування є способом створення програм, в яких єдиною дією є виклик функції, єдиним способом розбиття програми є створення нового імені функції та задання для цього імені виразу, що обчислює значення функції, а єдиним правилом композиції є оператор суперпозиції функцій. Функційне програмування не використовує жодних комірок пам’яті, інструкцій присвоєння, циклів, блок-схем чи передачі управління.

Структурне програмування – модель розроблення програмного забезпечення запропонована в 1970-х роках голландським науковцем Дейкстрою та була розроблена і доповнена Ніклаусом Віртом. Ґрунтується на теоремі Бьома-Якопіні, яка була опублікована у

1966 р. Згідно з цією моделлю будь-яка програма може бути створена використовуючи три конструкції:

• *послідовне виконання* – одноразове виконання операції в порядку запису їх (операцій) в тексті програми;

• *розгалуження* – виконання певної операції або декількох операцій в залежності від стану певної, наперед заданої умови;

• *цикл* – багаторазове виконання операції або групи операцій за умови виконання деякої наперед заданої умови. Така умова називається умовою продовження циклу.

Кожна конструкція являє собою блок із одним входом і одним виходом.

Об'єктно-орієнтоване програмування (ООП) – одна з парадигм програмування, яка розглядає програму як множину «об'єктів», що взаємодіють між собою. Основу ООП складають три основні концепції: інкапсуляція, успадкування та поліморфізм. Одною з переваг ООП є краща модульність програмного забезпечення (тисячу функцій процедурної мови, в ООП можна замінити кількома десятками класів із своїми методами). Попри те, що ця парадигма з'явилась в 1960-тих роках, вона не мала широкого застосування до 1990-тих, коли розвиток комп'ютерів та комп'ютерних мереж дозволив писати надзвичайно об'ємне і складне програмне забезпечення, що змусило переглянути підходи до написання програм. На даний час багато мов програмування або підтримують ООП або ж є цілком об'єктно-орієнтованими (зокрема, Java, C#, C++, Python, PHP, Ruby, Objective-C, ActionScript 3, Swift).

Компонентне програмування – це узагальнення ООП, орієнтоване на повторне використання програмних компонентів. Програмні компоненти незалежні від мови програмування і є реалізованими програмними об'єктами, які забезпечують виконання певної сукупності сервісів і подаються як контейнери з доступом до них через інтерфейс. Для інтеграції компонентів в кінцеві програми використовують крім контейнерів такі типові засоби як шаблони та каркаси. Компонентне програмування підтримується мовами ООП та системами RSEB, OOram, CORBA та ін.

Логічне програмування – це парадигма програмування, яка заснована на виведенні нових фактів з даних фактів згідно із заданими логічними правилами.

Проектування ПЗ – це процес визначення архітектури, компонентів, взаємозв'язків компонентів та їх інтерфейсів, а також інших функціональних характеристик системи. В любій моделі життєвого циклу проектування ПЗ складається з двох частин:

• архітектурного або високорівневого проектування – описується високорівнева структура підсистем і компонент, а також зв'язків між ними;

• компонентного або низькорівневого проектування – описується кожний компонент на рівні, достатньому для його реалізації.

Любу архітектуру можна подати з різних точок зору – поведінкової (динамічної), структурної (статичної), логічної (відповідність функціональним вимогам), фізичної (розподіленість), реалізації (подання у коді).

Відображення результатів проектування ПЗ у різних формах (візуальній, текстовій) називають нотаціями.

Для *структурного проектування* використовують наступні нотації:

• мови описання архітектури – тестові мови для описання архітектури в термінах компонентів і з'єднань;

• діаграми класів і об'єктів – описують набір класів і статичних зв'язків між ними;

• діаграми компонентів – аналогічні до діаграм класів, але подаються у графічній формі;

• карточки функціональної відповідальності і зв'язків класу – позначають імена класів і їх відповідальність (що вони мають роботи) і інші сутності (компоненти, актори/ролі);

• діаграми розгортання – використовуються для подання фізичних вузлів, зв'язків між ними і моделювання інших фізичних аспектів системи;

• діаграми сутність-зв'язок – використовуються для подання концептуальної моделі даних, яка зберігається в процесі роботи системи;

• мови описання/визначення інтерфейсу – подібні до мов програмування, але не мають можливостей описання логіки системи;

• структурні діаграми Джексона – описують структури даних у термінах послідовності, вибору і ітерацій;

• структурні схеми – описують структури виклику у програмах.

Для *поведінкового описання* використовують наступні нотації:

• діаграми діяльності або операцій – описують потоки робіт і керування;

• діаграми співробітництва – показують динамічну взаємодію у групі об’єктів (зв’язки і повідомлення);

• діаграми потоків даних – описують потоки даних всередині набору процесів;

• таблиці і діаграми прийняття рішень – подають складні комбінацій умов і дій;

• блок-схеми і структуровані блок схеми – подають потоки керування і зв’язані операції;

• діаграми послідовності – показують взаємодію всередині групи об’єктів з акцентом на часову послідовність повідомлень/викликів;

• діаграми переходів і карти станів – описують потоки керування переходами між станами;

• формальні мови специфікації – текстові мови, які використовують основні поняття з математики (наприклад, множини) для строгого і абстрактного визначення інтерфейсів і поведінки програмних компонентів в термінах перед і після-умов;

• псевдокод і програмні мови проектування – описують поведінку процедур і методів, в основному на стадії детального проектування;

При проектуванні використовуються наступні *стратегії*:

• “розділяй і володарюй” з покроковим уточненням;

• проектування “зверху-вниз” і “знизу-вверх”;

• абстракція даних і приховування інформації;

• ітеративний і інкрементний підхід.

На відміну від стратегій методи проектування більш специфічні і визначають свої внутрішні послідовності процесів Використовуються наступні *методи* проектування:

• Структурне – використовують для структурного аналізу з використанням діаграм потоків даних і зв’язаним описанням процесів.

• Функційно-орієнтоване – зосереджене на ідентифікації основних програмних функцій і уточненню цих функцій “зверху-вниз”.

• Об’єктно-орієнтоване – використовує поняття об’єкту (сутності), методу (дії), атрибуту (характеристики) і використовує інкапсуляцію, успадкування та поліморфізм.

• Проектування на основі структур даних – зосереджене на структурах даних, якими керує система, а не на функціях системи.

• Компонентне проектування - використовує незалежні одиниці, які мають однозначно визначені інтерфейси і залежності, та можуть збиратися і розгортатися незалежно одна від одної.

Для розв’язування типових проблем в процесі проектування можуть використовуватися шаблони проектування:

• Шаблони створення – `builder, factory, prototype, singleton`.

• Структурні шаблони – `adapter, bridge, composite, decorator, facade, flyweight, proxy`.

• Шаблони поведінки – `command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor`.

Наступним етапом після проектування є етап *реалізації*, на якому створюється робоче ПЗ шляхом комбінування кодування, верифікації (перевірки), модульного тестування, інтеграційного тестування і налагодження.

Технологія програмування (ТП) – це сукупність знань, методів і засобів які дозволяють організувати виробничий процес випуску ПЗ, в тому числі процеси планування, вимірювання характеристик, оцінки якості, відповідальності виконавця та ін. До цього часу наявне

протистояння двох позицій щодо поняття технологій програмування: з одного боку, під ним розуміють широке використання інструментальних засобів, а з іншого – технологія – це набір формальних методик та регламентів, які дозволяють на кожному етапі проводити експертизу, архівування та визначення обсягу та якості виконаної роботи. Перший варіант підтримують професійні програмісти, інший – керівники проектів.

Різні парадигми програмування відповідно використовують різні методології і технології програмування.

Технології зручно задавати у двох вимірах – вертикальному (процеси) і горизонтальному (стадії).

Процес – сукупність взаємозв'язаних дій, які перетворюють деякі вхідні дані у вихідні. Процеси складаються з набору *дій*, а кожна дія з набору *задач*. Вертикальний вимір відображає статичні аспекти процесів і оперує такими поняттями, як робочі процеси, дії, задачі, результати діяльності і виконавці.

Стадія – частина дії по створенню ПЗ, обмежена деякими часовими рамками і закінчується випуском конкретного продукту, визначеного завданнями для даної стадії. Стадії складаються з *етапів*, які звичайно мають ітераційний характер. Іноді стадії об'єднуються в більш великі часові рамки, які називають фазами. Горизонтальний вимір подається часом, відображає динамічні аспекти процесів і оперує такими поняттями, як фази, стадії, етапи, ітерації і контрольні точки.

Технологічний підхід визначається специфічною комбінацією стадій і процесів, орієнтованих на різні класи ПЗ і на особливості колективу розробників.

7. Авторські права і ліцензії на програмне забезпечення

При проектуванні і конструюванні ПЗ може використовуватися як вільне, так комерційне ПЗ, тому необхідно орієнтуватися в авторському праві і ліцензіях.

Авторське право (copyright) – це простий захист володіння певними видами інтелектуальної власності. Міжнародні домовленості про авторське право вимагають від учасників надавати авторським правам законну силу тільки у випадку:

"*… якщо з часу першої публікації всі копії роботи, опублікованої автором або іншим власником авторських прав, супроводжені фразою типу "Copyright © <ім'я власника авторських прав> <рік першої публікації>, розміщених таким чином, щоб звернути увагу на заяву про авторські права*".

Авторські права не вічні. Вся інтелектуальна власність з часом стає *загальним надбанням*. Це означає, що з часом суспільство присвоює авторські права на власність, і люба людина може роботи з цією власністю, що завгодно. Існує одна особливість: якщо створюється похідна робота, основана на роботі загального використання, то отримуються авторські права на *внесені зміни.*

Власники авторських прав можуть відкрито ставити **умови ліцензії.** Найбільш поширені області обмежень включають використання, копіювання, поширення і зміну.

Ліцензії на ПЗ діляться на дві великі групи:

- невільне (комерційне, пропрієтарне) і напіввільне ПЗ;
- вільне і відкрите ПЗ.

Основною характеристикою *комерційних ліцензій* є те, що видавець ПЗ в ліцензії дає дозвіл її отримувачу *використовувати* одну або більше копій програми, але сам при цьому залишається *правовласником* всіх цих копій.

Для комерційних версій характерно перераховувати велику кількість умов, які забороняють певні варіанти використання ПЗ. Прикладом комерційної ліцензії є ліцензія фірми Microsoft на ОС Windows, яка забороняє зворотну інженерію, одночасну роботу з системою декількох користувачів, поширення тестів її робочих характеристик, передачу ПЗ третім особам, а також передбачає можливість дистанційного блокування. Найбільш значним наслідком застосування комерційної ліцензії є те, що кінцевий користувач зобов'язаний її

прийняти, так як за законом власником ПЗ є не він, а видавець програми. Комерційне ПЗ поширюється згідно ліцензійної угоди з кінцевим користувачем (end user license agreememt, EULA) - це договір між власником програми і власником її копії.

Вільні і відкриті ліцензії не залишають права на конкретну копію програми її видавцю, а передають самі важливі частини з них кінцевому користувачу. В результаті користувач за замовчуванням отримує важливі права як *власника копії*, а всі *авторські права* на ПЗ залишаються у видавця.

Самою простою і історично першою є *ліцензія BSD*. Вона надає повну свободу поширенню коду на любих умовах, з сирцевими кодами або без них, і вимагає тільки збереження авторства. Нові розробники можуть тільки дописувати своє авторство.

Ліцензія GNU надає користувачеві права копіювати, змінювати та поширювати програми, а також зобов'язання, згідно з яким користувачі всіх похідних програм теж отримують вказані правила і зобов'язання. Принцип спадковості таких прав називають "*copyleft*".

Версія *GNU GPL v.1.0* є однією з перших обмежуючих ліцензій вільного ПЗ, яка вимагає:

- надання сирцевих кодів, доступних для вивчення, на додаток до бінарних кодів, які публікуються з цією версією;

- успадкування ліцензії у випадку модифікації сирцевого коду, тобто модифікований або об'єднаний з іншим код результату має бути випущений під ліцензією GNU GPL v.1.0.

Версія *GNU GPL v.2.0* водить обмеження на поширення ПЗ в державах, де кінцевий користувач не може в повній мірі використати свої права на модифікацію і поширення ПЗ під тією ж ліцензією.

Версія *GNU GPL v.2.1* призначена для бібліотек і дозволяє їх використання у комерційному ПЗ. Наприклад GNU C поширюється під цією ліцензією для того, щоб сторонні розробники могли використовувати їх у своєму вільному або комерційному ПЗ.

Версія *GNU GPL v.3* захищає користувачів і розробників від патентного переслідування, обмеження прав користувачів за допомогою технології Digital rights management (обмеження на копіювання і зміни вмісту), забороняє використовувати засоби технічного захисту, які протидіють внесенню змін користувачем у ПЗ, протидіє прив'язуванню ПЗ до обладнання.

Ліцензія *MIT/X* вимагає підтримки всіх існуючих повідомлень про авторські права і ліцензійні умови в сирцевому або двійковому поширюваному коді, і заборону використання імені любого автора без його письмової згоди з метою підтвердження або продовження похідних робіт.

Ліцензія *Artistic License* (творча ліцензія) зберігає права на необмежене відкрите поширення і запобігає продажу користувачем вдосконалених патентованих змін, які видають себе за офіційні версії.

Висновки.

• Системне ПЗ будується на наступних принципах: частотності, модульності, функціональної важливості і надлишковості, замовчуваності, переміщуваності, захисту, незалежності програм від зовнішніх пристроїв, сумісності.

• Системне ПЗ – це комплекс програмних засобів, які забезпечують керування компонентами як комп'ютерної системи (КС), такими як процесор, оперативна пам'ять, пристрої введення/виведення, мережеве обладнання, а також і задачами користувачів.

• Предметом дисципліни системного ПЗ є теорія і практика проектування, розроблення і функціонування системного ПЗ, яке забезпечує роботу обчислювальної системи.

• При розробленні системного ПЗ використовують мови програмування низького і високого рівня, а також мови написання сценаріїв.

• Для забезпечення переносимості системного ПЗ використовують інтерфейси API і ABI. Інтерфейс ABI дозволяє забезпечити сумісність частин ПЗ на рівні сирцевого коду, а інтерфейс ABI – на рівні двійкового коду.

• Для впорядкування процесу розроблення ПЗ створюються стандарти. Так POSIX стандарт забезпечує сумісність різних UNIX-подібних ОС та переносимості прикладних програм на рівні сирцевого коду.

• Під проектування ПЗ розуміють процес визначення архітектури, компонентів, взаємозв'язків компонентів та їх інтерфейсів, а також інших функціональних характеристик системи.

• Життєвий цикл розроблення ПЗ включає етапи планування, проектування, реалізації, модульного тестування, тестування підсистем, тестування системи, оціночне випробовування.

• Найбільш поширені моделі життєвого циклу розроблення ПЗ – класична послідовна, каскадна, спіральна, формалізована.

• Нотація – це відображення результатів проектування ПЗ у різних формах (візуальній, текстовій). Для структурного і поведінкового описання ПЗ використовують різні нотації.

• При проектуванні і конструюванні ПЗ необхідно дотримуватися авторських прав та ліцензій.

Література.

1. *Гордеев В.В., Молчанов А.Ю.* Системное программное обеспечение. – СПб.: Питер, 2001. – 736.
2. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.

Запитання.

1. Принципи побудови системне ПЗ?
2. Призначення системного ПЗ?
3. Предмет та задача системного ПЗ.
4. Місце системного ПЗ в структурі КС?
5. Засоби розроблення системного ПЗ?
6. Які програми відносяться до системних?
7. Яке призначення інтерфейсу прикладного програмування API?
8. Яке призначення інтерфейсу прикладного програмування AVI?
9. Для чого створюються стандарти на розроблення ПЗ?
10. Які етапи включає життєвий цикл ПЗ?
11. Які є моделі життєвих циклів та методології розроблення ПЗ?
12. Які парадигми використовуються при розробленні ПЗ?
13. Які нотації використовуються при проектуванні ПЗ?
14. Що описують процеси і стадії в технології програмування?
15. Що таке авторське право і які є ліцензії на ПЗ?

2. ОПЕРАЦІЙНІ СИСТЕМИ

Мета. Вивчення поняття, функцій, призначення, архітектури та переносимості операційних систем

Вступ. Операційна система забезпечує зв’язок між прикладними програмами й апаратними ресурсами комп’ютера. Операційна система приховує інтерфейс апаратного забезпечення, а пропонує інтерфейс прикладного програмування, який використовує абстракції вищого рівня. Виділення абстракцій дає змогу досягти незалежності прикладних програм від апаратного забезпечення. Основним завданням операційної системи є ефективний розподіл ресурсів обчислювальної системи, керування апаратним забезпеченням та організація виконання програм.

План.

1 Основи ОС
1.1 Поняття ОС
1.2 ОС, як розширена машина
1.3 ОС, як система керування ресурсами
1.4 Класифікація ОС
2. Сучасні ОС
2.1 Огляд сучасних ОС
2.2 Історія розвитку і основні характеристики ОС
3 Архітектура та структура ОС
3.1 Архітектуоа ОС
3.2 Мікроядерна архітектура
3.3 Екзоядро
3.4 Наноядро
4 Ядро та багатошарова структура ОС
4.1 Ядро та допоміжні модулі ОС
4.2 Ядро в привілейованому режимі
4.3 Багатошарова структура ОС
5 Апаратна залежність і переносимість операційних систем
5.1 Апаратна залежність ОС
5.2 Типові засоби апаратної підтримки ОС
5.3 Машинозалежні компоненти ОС
5.4 Переносимість ОС
6 Ресурси та їх класифікація

1 Основи ОС

1.1 Поняття операційної системи

Операційна система (*ОС*) – комплекс керувальних і оброблювальних програм, які виконують завдання керування ресурсами системи, й надають прикладним програмам операційне середовище для їх виконання.

Дві основні функції ОС:
– розширення можливостей ЕОМ;
– керування її ресурсами.

Операційне середовище – середовище виконання прикладних програм.
Операційне середовище визначає для прикладних програм множину команд процесора, які вони можуть використовувати, модель адресації й логічні структури адресного простору процесу,

множину доступних процесу системних викликів і т.ін. Операційна система може підтримувати декілька різних операційних середовищ.

Ресурси – повторно використовувані, відносно стабільні й часто відсутні об'єкти, які запитуються, використовуються й звільняються процесами в період їх активності.

Ресурс може бути *поділюваним*, коли декілька процесів можуть його використовувати одночасно (у той самий момент часу) або позмінно (на деякому інтервалі часу) і *неподілюваним*.

Існують *апаратні ресурси*, такі як процесорний час, оперативна пам'ять і дисковий простір; *програмні ресурси*, наприклад, бібліотеки функцій; *інформаційні ресурси* – вміст файлів і баз даних; *ресурси операційного середовища* – структури, використовувані для виконання системних викликів (наприклад, структура повідомлення); інші типи ресурсів. Для системи керування ресурсами поняття ресурсу зводиться до рівня абстрактної структури з набором атрибутів, які характеризують методи доступу до цієї структури і її фізичне подання в системі.

Одним з основних понять, пов'язаних з ОС, є поняття *процесу*.

Процес – абстракція, яка відображає базову одиницю обчислювальної роботи, що створюється ОС при запуску програми на виконання. Процес споживає різні ресурси ОС, наприклад:

− адресний простір процесу містить його програмний код, дані й стек (або стеки);
− файли використовуються процесом для зчитування вхідних даних і запису вихідних;
− обладнання введення-виведення використовується відповідно до його призначення.

Множина доступних процесу ресурсів і порядок їх використання визначаються архітектурою ОС. Зокрема, адресний простір процесу створюється в момент запуску програми на підставі інформації, отриманої ОС із вмісту програми й параметрів задання/запуску (якщо вони є). Залежно від архітектури обчислювального обладнання й ОС надаваний процесу адресний простір буде мати різні параметри (кількість адресних просторів, їх розмір, початкові й кінцеві адреси, способи адресації команд та даних і т.ін.).

Потік виконання або просто потік – абстракція, що являє собою виконання програми, яка розгортається в часі. Кожний процес має, принаймні, один потік. Потоки процесу спільно використовують його програмний код, глобальні змінні й системні ресурси, але кожний потік має власний програмний лічильник, власний вміст регістрів і власний стек. Процес є сукупністю взаємодіючих потоків і виділених для нього ресурсів.

1.2 ОС, як розширена машина

ОС приховує від програміста реальний стан апаратури й надає можливість простого і зручного перегляду файлів, зчитування або записування даних і т.ін. Так само, як ОС приховує від програмістів апаратуру дискового нагромаджувача і надає йому простий файловий інтерфейс, ОС обробляє переривання, керує таймерами й оперативною пам'яттю, а також виконує інші низькорівневі завдання. У кожному випадку та абстрактна, уявна машина, з якою завдяки ОС тепер може працювати користувач, набагато простіша й зручніша, ніж реальна апаратура, покладена в основу цієї абстрактної машини.

Отже, функцією ОС є надання користувачу деякої розширеної або віртуальної машини, яку легше програмувати і з якою простіше працювати, ніж безпосередньо з апаратурою, що становить реальну машину.

ОС приховує інтерфейс апаратного забезпечення, а пропонує інтерфейс прикладного програмування, який використовує абстракції вищого рівня.

Виділення абстракцій дає змогу досягти незалежності прикладних програм від апаратного забезпечення. Така характеристика системи називається апаратною незалежністю. Можна сказати, що ОС надає апаратно-незалежне середовище для виконання прикладних програм. Взаємодія ОС із апаратним забезпеченням показана на рис. 2.1.

Рисунок 2.1 – Взаємодія ОС із апаратним забезпеченням і прикладними програмами

1.3 ОС, як система керування ресурсами

Уявлення про те, що ОС насамперед система, яка забезпечує зручний інтерфейс користувачам, відповідає підходу «зверху вниз». Підхід «знизу вверх» дає уявлення про ОС як про деякий механізм, що керує всіма частинами складної системи. Сучасні обчислювальні системи складаються із процесорів, пам'яті, таймерів, дисків, мереж комунікаційної апаратури, друкарок і іншого обладнання. Відповідно до другого підходу функцією ОС є розподіл процесорів, пам'яті, обладнань і даних між процесами, що конкурують за ці ресурси.

ОС повинна керувати всіма ресурсами обчислювальної машини таким чином, щоб забезпечити максимальну ефективність її функціонування. Критерієм ефективності може бути, наприклад, пропускна здатність або реактивність системи.

До ресурсів відноситься все, що обчислювальна система може дати користувачу:
- процесорний час;
- місце в оперативній пам'яті;
- дисковий простір;
- засоби доступу до пристроїв введення/виведення;
- консоль (термінал з клавіатурою).

ОС розподіляє ці ресурси і надає їх на вимогу прикладним програмам.

Розрізняють два види розподілу ресурсів. У разі *просторового розподілу* ресурс доступний декільком споживачам одночасно, при цьому кожен із них може користуватися частиною ресурсу (так розподіляється пам'ять). При *часовому розподілі* система ставить споживачів у чергу і згідно з нею дає їм користуватися всім ресурсом обмежений час (так розподіляється процесор в однопроцесорних системах).

Керування ресурсами включає розв'язання двох загальних, незалежних від типу ресурсу, завдань:

• планування ресурсу – визначення кому, коли, а для поділюваних ресурсів – в якій кількості необхідно виділити цей ресурс;

• відстеження стану ресурсу – підтримання оперативної інформації про те, зайнятий чи не зайнятий ресурс, а для поділюваних ресурсів – яка кількість ресурсу вже розподілена, а яка вільна.

Для розв'язання цих загальних завдань керування ресурсами різні ОС використовують різні алгоритми, що і визначає їх вигляд в цілому, включаючи галузь застосування й користувацький інтерфейс. Так, наприклад, алгоритм керування процесором значною мірою визначає, чи є ОС системою поділу часу, системою пакетного оброблення чи системою реального часу.

1.4 Класифікація ОС

Операційні системи розрізняються особливостями реалізації внутрішніх алгоритмів керування основними ресурсами комп'ютера (процесорами, пам'яттю, обладнанням),

особливостями використаних методів проектування, типами апаратних платформ, галузями використання й багатьма іншими властивостями.

Особливості алгоритмів керування ресурсами. Залежно від особливостей використаного алгоритму керування процесором ОС поділяють на багатозадачні й однозадачні, багатокористувацькі й однокористувацькі, на системи, що підтримують багатониткове оброблення і що не підтримують його, на багатопроцесорні й однопроцесорні системи.

Підтримка багатозадачності. За кількістю одночасно виконуваних завдань ОС можна поділити на два класи: *однозадачні* (наприклад, MS-DOS) і *багатозадачні* (OS/2, UNIX, Linux, Windows та ін.).

Однозадачні ОС виконують здебільшого функцію надання користувачу віртуальної машини, роблячи більш простим і зручним процес взаємодії користувача з комп'ютером. Однозадачні ОС (рис. 2.2) розраховані на підтримку тільки одного процесу в кожний момент часу. Цей єдиний процес може мати тільки один потік. Програми можна запускати тільки послідовно – до завершення виконання процесу не можна створювати ще один процес. Однозадачні ОС включають в себе засоби керування периферійним обладнанням, засоби керування файлами, засоби спілкування з користувачем. У ході виконання процесу завдання однозадачної ОС зводиться до підтримки системних викликів.

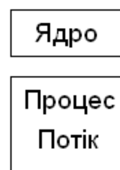

Рисунок 2.2 – Однозадачна ОС

Багатозадачні ОС підтримують одночасне існування декількох процесів, кожний з яких може мати тільки один потік (рис. 2.3). Багатозадачні ОС керують розподілом спільно використовуваних ресурсів, таких як процесор, оперативна пам'ять, файли й зовнішнє обладнання. В них завжди відбувається перехід виконання між потоками різних процесів, для чого потрібно перемикати контекст процесу й контекст потоку. Виконуваний потік змінює ядро, тобто перед зміною активного потоку відбувається перемикання в контекст ядра (змінюється при цьому контекст процесу або контекст потоку залежно від ОС).

Рисунок 2.3 – Декілька процесів (у кожного свій потік)

Багатозадачні операційні системи з підтримкою багатопоточності. У таких ОС до одного процесу можуть належати декілька потоків виконання команд (рис. 2.4). Усі потоки одного процесу розділяють його ресурси, наприклад, адресний простір або відкриті файли, проте характеризуються власним апаратним контекстом.

Рисунок 2.4 – Декілька процесів (у кожного – декілька потоків)

У такій системі може відбуватися перехід виконання від одного потоку процесу до іншого потоку того ж процесу. У цьому випадку не потрібно перемикати контекст процесу, відповідно таке перемикання проводиться швидше, ніж у випадку перемикання між процесами. Залежно від керування ресурсами власником або користувачем кожного типу ресурсів може бути або процес, або потік.

Підтримка багатокористувацького режиму. За кількістю одночасно працюючих користувачів ОС поділяють на однокористувацькі (MS-DOS, Windows 3.x) та багатокористувацькі (UNIX, Windows NT та ін.).

Головною відмінністю багатокористувацьких систем від однокористувацьких є наявність засобів захисту інформації кожного користувача від несанкціонованого доступу інших користувачів. Слід зазначити, що не кожна багатозадачна система є багатокористувацькою, і не кожна однокористувацька ОС є однозадачною.

Витісняльна і невитісняльна багатозадачність. Найважливішим розподілюваним ресурсом є процесорний час. Спосіб розподілу процесорного часу між декількома одночасно наявними в системі процесами (або нитками) багато в чому визначає специфіку ОС.

Серед варіантів реалізації багатозадачності можна виокремити дві групи алгоритмів:

– *невитісняльна багатозадачність* (Netware, Windows 3.x);

– *витісняльна багатозадачність* (Windows NT, OS/2, UNIX).

Основною відмінністю між варіантами витісняльної та невитісняльної багатозадачності є ступінь централізації механізму планування процесів. У першому випадку механізм планування процесів цілком зосереджений в ОС, а в другому – розподілений між ОС і прикладними програмами. За алгоритму невитісняльної багатозадачності активний процес виконується доти, поки він сам, за власною ініціативою, не віддасть керування ОС для того, щоб та вибрала з черги інший готовий до виконання процес. За алгоритму витісняльної багатозадачності рішення про перемикання процесора з одного процесу на інший, приймається ОС, а не активним процесом.

Підтримка багатонитковості. Важливою властивістю ОС є можливість розпаралелювання обчислень у межах одного завдання. *Багатониткова ОС* розподіляє процесорний час не між завданнями, а і між їхніми окремими гілками (нитками).

Багатопроцесорне оброблення. Загальноприйнятим є введення в ОС функцій підтримки багатопроцесорного оброблення даних. Такі функції є в ОС Solaris 2.x фірми Sun, Open Server 3.x компанії Santa Crus Operations, OS/2 фірми IBM, Windows NT фірми Microsoft і Netware 4.1 фірми Novell.

Багатопроцесорні ОС можна класифікувати за способом організації обчислювального процесу в системі з багатопроцесорною архітектурою: асиметричні ОС і симетричні ОС.

Асиметрична ОС виконується тільки на одному із процесорів системи, розподіляючи прикладні завдання по інших процесорах.

Симетрична ОС повністю децентралізована й використовує всі процесори, розподіляючи їх між системними й прикладними завданнями.

Вище були розглянуті характеристики ОС, пов’язані з керуванням тільки одним типом ресурсів – *процесором*. На вигляд ОС в цілому та на можливості її використання в тій або іншій

галузі впливають особливості й інших підсистем керування локальними ресурсами – підсистем керування пам’яттю, файлами, обладнанням введення-виведення.

Специфіка ОС проявляється й у тому, яким чином вона реалізовує мережеві функції: розпізнавання й перенаправлення у мережу запитів до вилучених ресурсів, передавання повідомлень по мережі, виконання вилучених запитів. У ході реалізації мережевих функцій виникає комплекс завдань, пов’язаних з розподіленим характером зберігання й оброблення даних у мережі: ведення довідкової інформації про всі доступні в мережі ресурси й сервери, адресація взаємодіючих процесів, забезпечення прозорості доступу, тиражування даних, узгодження копій, підтримка безпеки даних.

Особливості апаратних платформ. На властивості ОС безпосередньо впливають апаратні засоби, на які вона орієнтована. За типом апаратури розрізняють ОС персональних комп’ютерів (ПК), мінікомп’ютерів, мейнфреймів, кластерів та мереж комп’ютерів. Серед цих типів комп’ютерів можуть бути як однопроцесорні варіанти, так і багатопроцесорні. У кожному разі специфіка апаратних засобів звичайно відображається на специфіці ОС.

Очевидно, що ОС великої машини є більш складною й функціональною, ніж ОС ПК. Так, в ОС великих машин функції планування потоку виконуваних завдань, реалізуються шляхом використання складних пріоритетних дисциплін і потребують більшої обчислювальної потужності, ніж в ОС ПК. Аналогічно виконуються й інші функції.

Мережева ОС має у своєму складі засоби передавання повідомлень між комп’ютерами по лініях зв’язку, які не потрібні в автономній ОС. На основі цих повідомлень мережева ОС підтримує розподіл ресурсів комп’ютера між віддаленими користувачами, підключеними до мережі. Для підтримки функцій передавання повідомлень мережеві ОС містять спеціальні програмні компоненти, що реалізують популярні комунікаційні протоколи, такі як IP, IPX, Ethernet та ін.

ОС багатопроцесорних систем має особливу організацію, за допомогою якої сама ОС, а також підтримувані нею застосунки, могли б виконуватися паралельно окремими процесорами системи. Паралельна робота окремих частин ОС створює додаткові проблеми для розробників ОС, оскільки в цьому випадку набагато складніше забезпечити погоджений доступ окремих процесів до загальних системних таблиць, виключити ефект змагань та інші небажані наслідки асинхронного виконання робіт.

Інші вимоги ставляться до ОС кластерів.

Кластер – слабко поєднана сукупність декількох обчислювальних систем, що працюють спільно для виконання спільних застосунків, що подаються користувачу єдиною системою.

Поряд зі спеціальною апаратурою для функціонування кластерних систем необхідна й програмна підтримка з боку ОС, яка зводиться в основному до синхронізації доступу до розподілюваних ресурсів, виявлення відмов і динамічної реконфігурації системи. Однією з перших розробок у галузі кластерних технологій були рішення компанії Digital Equipment на базі комп’ютерів VAX. Ця компанія уклала угоду з корпорацією Microsoft про розроблення кластерної технології, що використовує Windows NT. Кілька компаній пропонують кластери на основі Unix-Машин.

Поряд з ОС, орієнтованими на певний тип апаратної платформи, існують ОС, спеціально розроблені таким чином, щоб їх можна було легко переносити з комп’ютера одного типу на комп’ютер іншого типу – мобільні ОС. Найбільш яскравим прикладом такої ОС є популярні системи UNIX і Linux. У цих системах апаратнозалежні місця ретельно локалізовані, тому під час перенесення системи на нову платформу переписуються тільки вони. Засобом, що полегшує перенесення іншої частини ОС, є написання її машинно-незалежною мовою, наприклад, С, яку і було розроблено для програмування ОС.

Особливості галузей використання. Багатозадачні ОС підрозділяють на три типи відповідно до використовуваних для їх розроблення критеріїв ефективності:

– *системи пакетного оброблення* (ОС ЕС);

– *системи розділення часу* (UNIX, VMS);

– *системи реального часу* (QNX, RT/11).

Системи пакетного оброблення призначались насамперед для розв'язання обчислювальних задач, що не потребують швидкого отримання результатів. Головною метою й критерієм ефективності систем пакетного оброблення є максимальна пропускна здатність, тобто розв'язування максимальної кількості завдань за одиницю часу. Для досягнення цієї мети в системах пакетного оброблення використовується така схема функціонування: на початку роботи формується пакет завдань, кожне завдання містить вимогу до системних ресурсів; із цього пакета завдань формується мультипрограмна суміш, тобто множина одночасно виконуваних завдань. Для одночасного виконання вибираються завдання, що потребують різних ресурсів для забезпечення збалансованого завантаження всіх блоків обчислювальної машини; так, наприклад, у мультипрограмній суміші бажана одночасна наявність обчислювальних завдань і завдань із інтенсивним введенням-виведенням. Таким чином, вибір нового завдання з пакета завдань залежить від внутрішньої ситуації, що складається в системі, тобто вибирається «вигідне» завдання. Отже, у таких ОС неможливо гарантувати виконання того або іншого завдання протягом певного періоду часу. У системах пакетного оброблення процесор з виконання одного завдання перемикається на виконання іншого тільки в тому випадку, якщо активне завдання саме відмовляється від процесора, наприклад, через необхідність виконання операції введення-виведення. Тому одне завдання може надовго зайняти процесор, що робить неможливим виконання інтерактивних завдань. Таким чином, взаємодія користувача з обчислювальною машиною, на якій встановлено систему пакетного оброблення, зводиться до того, що він завдання подає диспетчеру-оператору, а після виконання всього пакета завдань отримує результат. Очевидно, що такий порядок знижує ефективність роботи користувача.

В *системах розділення часу* усунуто основний недолік систем пакетного оброблення – ізоляція користувача-програміста від процесу виконання його завдань. Кожному користувачу системи розділення часу надається термінал, з якого він може вести діалог зі своєю програмою. Оскільки в системах розділення часу для кожного завдання виділяється тільки квант процесорного часу, жодне завдання не займає процесор надовго, і час відповіді виявляється прийнятним. Якщо квант обирається досить малим, то всі користувачі, що одночасно працюють на одній і тій же машині, вважають, що кожен з них одноосібно використовує машину. Зрозуміло, що системи розділення часу мають меншу пропускну здатність, ніж системи пакетного оброблення, тому що до виконання приймається кожне запущене користувачем завдання, а не те, яке «вигідне» системі, і, крім того, є накладні витрати обчислювальної потужності на більш часте перемикання процесора із завдання на завдання. Критерієм ефективності систем розділення часу є не максимальна пропускна здатність, а зручність і ефективність роботи користувача.

Системи реального часу застосовуються для керування різними технічними об'єктами, такими як, наприклад, супутник, наукова експериментальна установка, або технологічними процесами, такими, як гальванічна лінія, доменний процес і т.ін. Для цих випадків існує гранично допустимий час, протягом якого має бути виконана та або інша програма, що керує об'єктом, а якщо ні, то може статися аварія: супутник вийде із зони видимості, експериментальні дані, що надходять з давачів, будуть загублені, товщина гальванічного покриття не буде відповідати нормі. Таким чином, критерієм ефективності для систем реального часу є їх здатність витримувати заздалегідь задані інтервали часу між запуском програми й отриманням результату (керувальний вплив). Цей час називають часом реакції системи, а відповідна властивість системи – реактивністю. Для цих систем мультипрограмна суміш являє собою фіксований набір заздалегідь розроблених програм, а програма до виконання вибирається виходячи з поточного стану об'єкта або відповідно до розкладу планових робіт.

Деякі ОС можуть поєднувати в собі властивості систем різних типів, наприклад, частину завдань можна виконувати в режимі пакетного оброблення, а частину – у режимі реального часу або в режимі розділення часу. В таких випадках режим пакетного оброблення часто називають фоновим режимом.

Особливості методів побудови. Описуючи ОС, часто вказують особливості її структурної організації й основні концепції, покладені в її основу. Розглянемо такі базові концепції.

Способи побудови ядра системи – монолітне ядро або мікроядерний підхід. Більшість ОС використовує монолітне ядро, яке компонується як одна програма, що працює в привілейованому режимі і використовує швидкі переходи з однієї процедури на іншу, які не вимагають перемикання з привілейованого режиму в користувацький і навпаки. Альтернативою є побудова ОС на базі мікроядра, що працює також у привілейованому режимі й виконує тільки мінімум функцій з керування апаратурою, в той час, як функції ОС більш високого рівня виконують спеціалізовані компоненти ОС – сервери, що працюють у користувацькому режимі. За такої побудові ОС працює більш повільно, оскільки часто виконуються переходи між привілейованим режимом і користувацьким, зате система виходить більш гнучкою – її функції можна нарощувати, модифікувати або звужувати, крім серверів користувацького режиму. Крім того, сервери добре захищені один від одного, як і будь-які користувацькі процеси.

Побудова ОС на базі об'єктно-орієнтованого підходу дає можливість використовувати всі його переваги, що добре зарекомендували себе на рівні застосунків, усередині ОС, а саме: акумуляцію вдалих рішень у формі стандартних об'єктів, можливість створення нових об'єктів на базі наявних за допомогою механізму успадкування, захист даних за рахунок їх інкапсуляції у внутрішні структури об'єкта, що робить дані недоступними для несанкціонованого використання ззовні, структурованість системи, що складається з набору добре визначених об'єктів.

Наявність декількох прикладних середовищ дає змогу у межах однієї ОС одночасно виконувати застосунки, розроблені для кількох ОС. Багато сучасних ОС підтримують одночасно прикладні середовища MS-DOS, Windows, UNIX, Linux (POSIX), OS/2 або хоча б деяку підмножину із цього набору. Концепція множинних прикладних середовищ просто реалізується в ОС на базі мікроядра, з яким працюють різні сервери, частина яких реалізовує прикладне середовище тієї або іншої ОС.

Розподілена організація ОС дозволяє спростити роботу користувачів і програмістів у мережевих середовищах. У розподіленій ОС реалізовано механізми, які дають можливість користувачу уявляти й сприймати мережу як традиційний однопроцесорний комп'ютер. Характерними ознаками розподіленої організації ОС є: наявність єдиної довідкової служби поділюваних ресурсів, єдиної служби часу, використання механізму виклику віддалених процедур (RPC) для прозорого розподілу програмних процедур по машинах, багатониткового оброблення, що дозволяє розпаралелювати обчислення в межах одного завдання і виконувати це завдання відразу на декількох комп'ютерах мережі, а також наявність інших розподілених служб.

2 Сучасні ОС

2.1 Огляд сучасних ОС

Операційні системи мейнфреймів. На верхньому рівні міститься ОС для мейнфреймів. Ці комп'ютери розміром з кімнату все ще можна знайти в центрах даних великих корпорацій. Мейнфрейми відрізняються від ПК можливостями введення-виведення. Досить часто трапляються мейнфрейми з тисячами дисків і терабайтами даних. Мейнфрейми немовби повертаються у вигляді потужних web-серверів, серверів для великомасштабних електронно-комерційних сайтів і серверів для транзакцій в бізнесі.

Операційні системи для мейнфреймів насамперед орієнтовані на оброблення одночасно множин завдань, більшість з яких потребує численних операцій введення-виведення. Звичайно вони пропонують три види обслуговування: пакетне оброблення, оброблення транзакцій (групові операції) й розділення часу. Пакетне оброблення є системою, що виконує стандартні завдання без присутності користувачів, які працюють в інтерактивному режимі. Оброблення

позовів в страхових компаніях або складання звітів про продажі для мережі – це типові завдання, що обробляються в пакетному режимі. Системи оброблення транзакцій керують дуже великою кількістю дрібних запитів, наприклад контролюють процес роботи в банку або бронювання авіаквитків. Кожен окремий запит невеликий, але система повинна відповідати на сотні або тисячі запитів за секунду. Системи розділення часу, дозволяють великій кількості віддалених користувачів одночасно виконувати завдання на одній машині. Наочним прикладом є робота з великою базою даних. Всі ці функції тісно пов'язані між собою, і часто ОС мейнфрейму виконує їх. Прикладом ОС для мейнфрейму є OS/390 на базі OS/360.

Серверні операційні системи. Рівнем нижче розміщено серверні ОС. Вони працюють на серверах, які є або дуже великими ПК, або робочими станціями, або навіть мейнфреймами. Вони одночасно обслуговують багатьох користувачів і дозволяють їм ділити між собою програмні та апаратні ресурси. Сервери дають змогу працювати з пристроями друку, файлами або з Інтернетом. Інтернет провайдери зазвичай запускають у роботу декілька серверів для підтримання одночасного доступу до мережі безлічі клієнтів. На серверах зберігаються сторінки web-сайтів і обробляються вхідні запити. UNIX, Linux і Windows є типовими серверними ОС.

Багатопроцесорні операційні системи. Найбільшого поширення набув спосіб збільшення потужності комп'ютерів, що полягає в з'єднанні декількох центральних процесорів в одній системі. Залежно від виду з'єднання процесорів і розділення роботи такі системи називаються паралельними комп'ютерами, мультикомп'ютерами або багатопроцесорними системами. Вони потребують спеціальних ОС, але часто такі ОС є варіантами серверних ОС із спеціальними можливостями зв'язку.

Операційні системи для персональних комп'ютерів. Операційні системи для ПК надають зручний інтерфейс для одного користувача. Такі системи широко використовують для роботи з текстом, електронними таблицями і доступу до Інтернету. Найбільш поширені – це Windows 98, Windows 2000, Windows 7, ОС комп'ютера Macintosh і Linux. Насправді багато людей навіть не знають про існування інших видів ОС, окрім тієї, якою вони користуються.

Операційні системи реального часу. Ще один вид ОС – це *системи реального часу*. Головним параметром таких систем є час. Часто такі процеси мають задовольняти жорсткі часові вимоги. Якщо деяка дія має відбутися в конкретний момент часу (або в заданому діапазоні часу), це буде *жорстка система реального часу*.

Існує також *гнучка система реального часу*, для якої допустимі пропуски термінів виконання операції, що трапляються час від часу. Це зокрема цифрові аудіо- і мультимедійні системи. Системи Vx Works і QNX є добре відомими ОС реального часу.

Вбудовані операційні системи. Наступним кроком від величезних систем до менших є кишенькові комп'ютери і вбудовані системи. Кишеньковий комп'ютер або PDA (Personal Digital Assistant – персональний цифровий помічник) – це малий за розміром комп'ютер, що виконує невеликий набір функцій (телефонного записника та блокнота). Вбудовані системи, призначені для керування діями пристроїв, працюють на машинах, що звичайно не вважаються комп'ютерами, наприклад в телевізорах, мікрохвильових печах і мобільних телефонах. Їх характеристики часто такі самі, що й систем реального часу, але при цьому вони мають особливий розмір, пам'ять і обмеження потужності, що виділяє їх в окремий клас. Прикладами таких ОС є PALM OS і Windows CE (Consumer Electronics – побутова техніка).

Операційні системи для смарт-карт. Найменші ОС працюють на смарт-картах, які є пристроями розміром з кредитну карту, що містять центральний процесор. На такі ОС накладаються жорсткі обмеження потужності процесора і пам'яті. Деякі з них можуть керувати тільки однією операцією, наприклад електронним платежем, але інші ОС на тих же самих смарт-картах виконують складні функції. Часто вони є патентованими системами.

Деякі смарт-карти є Java-орієнтованими. Це означає, що постійний запам'ятовуючий пристрій (ПЗП, або ROM — Read Only Memory – пам'ять тільки для читання) смарт-карт містить інтерпретатор віртуальної машини Java (JVM – Java Virtual Machine). Аплети Java (маленькі програми) завантажуються на карту і виконуються JVM-інтерпретатором. Деякі з

таких карт можуть одночасно керувати декількома аплетами Java, що приводить до багатозадачності й необхідності планування. За одночасної роботи двох і більше програм виникає потреба в керуванні ресурсами і захистом. Відповідно всі ці завдання виконує примітивна ОС, розміщена у смарт-карті.

2.2 Історія розвитку і основні характеристики ОС

Операційна система Unix. Системи Unix розроблялися різними виробниками, тому доцільно розглянути історію створення сім’ї цих ОС. У 1968 р. консорціум дослідників фірм General Electric, AT&T Bell Labs і Массачусетського технологічного інституту завершив роботу над науково-дослідним проектом Multics, результатом якого стала операційна система, яка увібрала до свого складу останні досягнення у вирішенні проблем багатозадачності, керування файлами та взаємодії з користувачем. У 1969 р. Кен Томпсон розробив ОС Unix, у якій використано багато результатів проекту Multics. Він пристосував цю систему, призначену для роботи на міні-ЕОМ, до потреб дослідників. Із самого початку Unix стала зручною, ефективною, розрахованою на багато користувачів і багатозадачною ОС.

З часом популярність Unix в Bell Labs зростала, і в 1970 р. Денніс Рітчі і Кен Томпсон переписали код системи мовою програмування C. Денніс Рітчі, колега Томпсона з Bell Labs, створив цю мову для забезпечення гнучкості під час розроблення програм. Одна з переваг мови C полягає в тому, що вона дозволяє звертатися безпосередньо до апаратних засобів комп’ютера за допомогою узагальненого набору команд. До цього текст програми ОС потрібно було спеціально переписувати апаратно-залежною мовою Assembler для кожного типу комп’ютера. Мова C дозволила Рітчі та Томпсону написати всього одну версію ОС Unix, яку потім можна було компілювати C-компіляторами на різних машинах. Операційна система Unix стала мобільною, тобто здатною працювати на різних типах машин без перепрограмування.

Поступово Unix стала стандартним програмним продуктом, який поширювався багатьма фірмами, включаючи Novell та IBM. Спочатку цю ОС вважали дослідним продуктом, тому перші версії розповсюджувалися безкоштовно по факультетах обчислювальної техніки багатьох відомих університетів. У 1972 р. Bell Labs почала випускати офіційні версії Unix і продавати ліцензії на неї різним користувачам. Одним з таких користувачів був факультет обчислювальної техніки Каліфорнійського університету в Берклі. Його фахівці ввели в систему багато нових особливостей, які згодом стали стандартними. У 1975 р. у Берклі була випущена власна версія системи, відома як Berkeley Software Distribution (BSD). Ця версія Unix стала основним суперником версії AT. За нею послідувала System V, яка стала важливим підтримуваним програмним продуктом. Паралельно випускалися версії BSD. Наприкінці 70-х років BSD Unix стала основою дослідницького проекту, що виконувався в Агентстві перспективних досліджень і розробок (DARPA) міністерства оборони США. У результаті в 1983 р. Каліфорнійський університет випустив потужну версію системи під назвою BSD 4.2. Вона включала в себе досить досконалу систему керування файлами і мережеві засоби, засновані на використанні протоколів TCP/IP, що застосовуються в Інтернеті. Версія BSD 4.2 набула поширення і була обрана багатьма фірмами-виробниками, зокрема Sun Microsystems.

Поширення різних версій Unix зумовило потребу у розробленні стандарту на цю ОС. Іншого способу дізнаватися про те, в яких версіях будуть працювати призначені для використання в цьому середовищі програми, у розробників ПЗ не було. В середині 80-х років з’явилися два конкуруючі стандарти: один був створений на основі версії AT. У 1991 р. Unix System Laboratories розробила System V версії 4, в якій реалізовано майже всі можливості варіантів попередньої версії BSD версії 4.3, SunOS і Xenix. У відповідь кілька компаній, зокрема, IBM і Hewlett-Packard, створили фонд відкритого програмного забезпечення (Open Software Foundation, OSF), метою якого стало розроблення власної стандартної версії Unix. У результаті з’явилися два конкуруючі комерційні стандартні варіанти: версія OSF і System V версії 4. У 1993 р. компанія AT&T продала свою частку прав на Unix фірмі Novell, і деякий час Unix Systems Laboratories належала до Novell. За цей час фірма випустила власні версії Unix на

базі System V версії 4 під загальною назвою UnixWare, призначені для взаємодії із системою NetWare розробки Novell.

Протягом свого розвитку Unix залишалася великою і вимогливою до апаратних засобів ОС, для ефективної роботи якої необхідна робоча станція або міні-ЕОМ. Деякі версії ОС були розраховані в основному на робочі станції. Те, що ця ОС встановлюється на комп'ютерах всіх типів (робочих станціях, міні-ЕОМ і навіть супер-ЕОМ), є свідченням її мобільності, що забезпечила можливість ефективної версії Unix для ПК.

Операційна система Linux. Найпоширенішим проектом системи Unix кінця XX ст. стала альтернатива дорогим рішенням – ОС Linux. Тепер темпи освоєння ринку цією системою найбільш інтенсивні порівняно з іншими відомими ОС.

Створення цієї системи починалося з розроблення проекту Лінуса Торвальда – студента факультету обчислювальної техніки Гельсинкського університету. У той час студенти користувалися програмою Minix, яка демонструвала різні можливості Unix. Ця програма, розроблена професором Ендрю Таннебаумом, поширилася по мережі Інтернет серед студентів усього світу.

Лінус поставив за мету створити ефективну ПК-версію Unix для користувачів Minix. Він назвав її Linux і в 1991 р. випустив версію 0.11. Система широко розповсюдилася по Інтернету і в наступні роки була доопрацьована іншими програмістами, які ввели до неї можливості та особливості, притаманні стандартним системам Unix.

Зокрема, було перенесено всі основні програми-менеджери вікон. У цій ОС використовуються утиліти Інтернет, є і повний набір засобів розроблення програм, включаючи компілятори і налагоджувач C. Незважаючи на такі широкі можливості, ОС Linux залишається невеликою, стабільною й швидкодійною. У мінімальній конфігурації вона може ефективно працювати навіть на 386 комп'ютерах за ємності оперативної пам'яті 4 Мбайт.

Сильною стороною Linux є її універсальність. Система покриває весь діапазон застосувань: від настільного ПК до надпотужних багатопроцесорних серверів і кластерів.

Linux виконує ті ж функції, що й DOS і Windows, однак відрізняється від них особливою потужністю й гнучкістю. Більшість ОС ПК створювалися для невеликих ПК, що мали обмежені можливості і лише нещодавно перетворилися на універсальні машини. Такі ОС постійно модернізуються, щоб відповідати можливостям апаратних засобів ПК, які безперервно розвиваються. Linux же розроблялася в зовсім іншому контексті.

Розроблення початкової для Linux системи Unix полягало в створенні продукту, який міг би задовольняти співробітників, що займаються різними дослідженнями. Операційна система розглядалася як механізм, що надає користувачу набір високоефективних інструментів. Така орієнтація на користувача означала можливість конфігурації і програмування системи відповідно до конкретних потреб. У випадку з Linux ОС дійсно стала операційним середовищем.

З фінансового погляду Linux має одну істотну перевагу: вона є не комерційною, і на відміну від ОС Unix поширюється за генеральною відкритою ліцензією GNU в межах фонду безкоштовного ПЗ, тому ця ОС доступна для всіх. GNU складена таким чином, що Linux залишається безкоштовною і водночас стандартизованою системою – існує лише один офіційний її варіант.

Апаратні потреби для Linux — мінімальні (рекомендовані):

– пам'ять: 4 Мбайт (32 Мбайт);

– процесор: 80386 (IP-166 МГц) або сумісний;

– вінчестер: 100 Мбайт (600 Мбайт).

Операційні системи сім'ї Windows. *Перші версії Windows*. Перша версія Windows вийшла в світ наприкінці 80-х років і залишилася абсолютно непоміченою. Аналогічна доля спіткала і наступну версію – лише версія Windows 3.0 (1992) зуміла прокласти собі дорогу і стати «продуктом року». А ще через два роки були випущені версії 3.1 і 3.11, які остаточно укріпили динамічні позиції Windows. Остання включала повну підтримку мультимедіа і роботу в локальній мережі – тому й отримала уточнюючу назву Windows For Workgroups.

Апаратні потреби для Windows 3.1 – мінімальні (рекомендовані):
– пам’ять: 1 Мбайт (4 Мбайт);
– процесор: 80286 (80386) або сумісний;
– вінчестер: 20 Мбайт (80 Мбайт).
Апаратні потреби для Windows 3.11 – мінімальні (рекомендовані):
– пам’ять: 2 Мбайт (8 Мбайт);
– процесор: 80386 (80486) або сумісний;
– вінчестер: 40 Мбайт (100 Мбайт).

Покоління 9Х. Windows 95. Нова ОС, мала була вийти ще в 1994 р. – саме тоді з’явилися офіційні повідомлення про завершення розроблення нової ОС, що отримала назву Chicago. Однак термін представлення «Чикаго» постійно відкладався, корпорація Microsoft робила щоразу обнадійливі заяви. Зрештою у серпні 1995 р. Windows 95 вийшла в світ.

Більше того – нова ОС стала 32-розрядною. Усі попередні версії DOS і Windows були 16-розрядними і, отже, не могли повною мірою використовувати можливості навіть процесорів родини 386 і, тим паче, нових процесорів Pentium. Звичайно в цьому полягали й деякі незручності – спеціально під Windows 95 користувачам довелося замінювати Windows-програми на нові 32-розрядні версії. Однак на практиці перехід виявився порівняно легким – уже за рік з’явилися нові версії програмних продуктів.

Windows 95 отримала абсолютно новий графічний інтерфейс – більш елегантний, зручний для користувача і зовні привабливий порівняно з попередніми ОС.

Windows 98 і Windows 98SE. До роботи над новою версією Windows корпорація Microsoft приступила відразу ж після виходу Windows 95. Нова ОС очікувалася наприкінці 1996 р. і мала називатися Memphis. Але цього не сталося ні в 1996 р., ні в 1997 р. Тільки 25 червня 1998 р. нова ОС Microsoft надійшла до магазинів.

Основні зміни торкнулися інтерфейсу – тепер «Робочий стіл» Windows 98 став ще красивішим, а головне – він повністю інтегрований із середовищем Інтернет. У новій ОС остаточно була стерта відмінність між файлами і каталогами на комп’ютері та об’єктами Всесвітньої інформаційної павутини (WorldWideWeb). Основний засіб роботи з файлами та каталогами в обох випадках – програма Internet Explorer.

Інша важлива відмінність Windows 98 від Windows 95 полягає в розширених можливостях керування інтерфейсом. Але є і більш важливі зміни – у внутрішній будові ОС. Хоча основна «начинка» ОС залишилася колишньою, Windows 98 вигравала у попередньої ОС за рахунок коректної роботи з новими комплектуючими – процесором Pentium II, графічним портом AGP, шиною USB, новими моделями відеокарт, материнських плат, модемів і т. ін. Нарешті Windows 98 містила велику кількість нових програм і утиліт – в першу чергу повний комплект ПЗ для роботи в Інтернеті та утиліту конвертації файлової системи FAT16 у більш нову версію FAT32.

Наприкінці 1999 р. у з’явилася версія нового комплекту Windows 98 – Windows 98 SE. Від попередньої версії нова Windows відрізнялась тим, що до її складу включено п’яту версію переглядача Internet Explorer, оновлено систему з’єднання з Інтернетом, а також зроблено численні виправлення помилок і є нова бібліотека драйверів пристроїв.

Windows ME (Microsoft Windows Millennium Edition) – остання еволюція ОС класу Windows 95 – Windows 98, запущена в серійне виробництво в 2000 р.

Windows ME значно відрізнялась від родини системних платформ Windows 9X, передусім тим, що в цій реалізації Windows зовсім не підтримується MS DOS – коректно запустити на комп’ютері, що працює під керуванням цієї системи, деякі програми DOS – досить складне завдання. Windows ME тісно інтегрована з Internet Explorer 5.0, що зробило її ще більше ресурсомісткою, в комплект поставки за замовчуванням включено більшу частину елементів Microsoft Plus для Windows 98, базовий набір ігор розширено новими програмами, що дозволяють користувачу «грати» в мережі Інтернет з живими суперниками, додано Windows Media Player 7.0, що підтримує відтворення файлів багатьох нових аудіо- та відеоформатів. Інтерфейс Windows ME майже повністю збігається із зовнішнім оформленням Windows 2000

Professional, включаючи системні іконки й оновлене діалогове вікно вимкнення/перезавантаження комп'ютера, але майже всі базові елементи налаштування Windows 98 збереглися на своїх колишніх місцях. Windows ME дійсно стала останньою ОС сім'ї Windows 9X, оскільки всі наступні ОС Windows як для домашніх комп'ютерів, так і для робочих станцій, створювалися на платформі NT.

Апаратні потреби для Windows ME – мінімальні (рекомендовані):

– пам'ять: 32 Мбайт (64 Мбайт);

– вінчестер: 500 Мбайт.

Покоління NT. Windows NT (New Technology). 32-розрядна Windows NT, перша версія якої з'явилася на ринку в 1993-му, а остання – у 1998 р., із самого початку створювалася як надстабільна, надійна система, розрахована передусім на роботу. І в цьому сенсі Windows 98/ME значно їй уступала, так як випадки помилок, крахів і «зависання» під час роботи у Windows NT траплялися вкрай рідко. Відбувалося це тому, що у Windows NT розроблено надійне розділення програм, які працюють під її керуванням, що не дає їм «змагатися» за ресурси. У Windows 3.1/95/98/ME кожна із завантажених програм була незалежною від інших. Нерідко програми перезавантажували процесор запитами на ресурси, у результаті чого ОС «зависала».

На відміну від Windows 98/ME Windows NT забороняє беззаперечний доступ до ресурсів комп'ютера будь-яким програмам. Це дозволяє системі уникнути конфліктів, але в результаті під NT не працювали програми, написані для DOS, і багато створених для Windows 95.

Слід враховувати і той факт, що велика частина роботи виконується з NT лише в мережевому режимі роботи, тобто разом з іншими комп'ютерами.

Windows 2000. Вона з'явилася на ринку на початку 2000 р. Операційна система Microsoft Windows 2000 являє собою друге покоління ОС, побудованих за архітектурою Windows NT. Вона випускається в трьох модифікаціях: Windows 2000 Professional для ноутбуків, настільних систем і робочих станцій, Windows Server 2000 для серверних комп'ютерів і Windows 2000 Datacenter Server для великих серверних систем, робочих станцій великих корпоративних мереж та спеціалізованих банківських і файлових серверів.

Завдяки використанню вдосконаленої технології NT, що поєднується з об'єктивною простотою інтерфейсу Windows 9X, Windows 2000 має високу надійність і стабільність, також вона значно легше піддається налаштуванню та конфігурації, ніж попередні версії Windows. Розмежування доступу до системи реалізовано на високому рівні, що дозволяє забезпечити безпеку збереження даних на дисках, якщо за комп'ютером працює більше ніж один користувач. Windows 2000 була визнана однією з найкращих, і досі використовується на багатьох комп'ютерах, незважаючи на вихід більш нових версій ОС Windows.

Windows XP. Операційна система Microsoft Windows XP (від англ. EXPerience – досвід) відома також під кодовим найменуванням Microsoft Codename Whistler. Спочатку корпорація Microsoft планувала розробити дві незалежні ОС нового покоління. Перший проект отримав робочу назву Neptune, ця ОС мала б стати черговим оновленням Windows ME, новою системою лінійки Windows 9X. Другий проект, що мав назву Odyssey, передбачав створення ОС на платформі Windows NT, яка повинна змінити Windows 2000. Проте керівництво Microsoft визнало недоцільним розосереджувати ресурси на просування двох різних ОС, унаслідок чого обидва напрями розробок були об'єднані в один проект – Microsoft Whistler. Можливо, саме завдяки цьому Windows XP поєднує в собі переваги ОС попередніх поколінь: зручність, простоту в інсталяції та експлуатації ОС сім'ї Windows 98 і Windows ME, а також надійність і багатофункціональність Windows 2000. Windows XP для настільних ПК і робочих станцій випускалася в трьох модифікаціях: Home Edition для домашніх ПК, Professional Edition – для офісних ПК і, нарешті, Microsoft Windows XP 64bit Edition – це версія Windows XP Professional для ПК, складених на базі 64-бітного процесора Intel Itanium з тактовою частотою понад 1 ГГц.

Апаратні потреби для Windows XP, мінімальні:

– пам'ять: 64 Мбайт;

– процесор: Pentium – сумісний, тактова частота від 233 МГц;

– вільний дисковий простір: 1,5 Гбайт.

Windows.NET. Операційна система MS Windows.NET – це родина серверних ОС, розроблених корпорацією Microsoft на основі Windows XP, які змінили Windows 2000 Server, Advanced Server і Datacenter Server. Windows.NET поставляється у варіантах Windows.NET Server, Windows.NET Advanced Server і Windows.NET Datacenter Server. Відповідно технічні можливості цих версій ОС розрізняються: наприклад, Windows.NET Server може адресувати чотирипроцесорні системи, Windows.NET Advanced Server працює з восьмипроцесорними комп'ютерами, а Windows.NET Datacenter Server підтримує машини, апаратна конфігурація яких включає до 32 синхронно працюючих процесорів.

Windows Vista. Ця версія Windows вийшла восени 2006 р. Усього випущено сім варіантів Windows Vista, які можна розбити на дві групи – Home і Business.

Windows 7. Компанія Microsoft випустила нову ОС Windows 7. У Windows 7 є можливість вимкнення або ввімкнення переглядача Internet Explorer і програвача Windows Media Player. Також ОС має підтримку multitouch-моніторів.

Функція Branch Cache дозволяє зменшити затримки у користувачів, що працюють з комп'ютером віддалено. Наприклад, файл доступний в мережі, кешується локально, тому він скачується вже не з віддаленого сервера, а з локального комп'ютера. Ця функція може працювати в двох режимах – Hosted Cache і Distributed Cache. У першому випадку файл зберігається на виділеному локальному сервері під керуванням Windows Server 2008 R2, у другому – на комп'ютері у клієнта.

Функція ReadyBoost дозволяє використовувати флеш-нагромаджувач як додаткову кеш-пам'ять для прискорення роботи системи.

Windows 10. Це остання версія Windows вийшла осінню 2014 року. Windows 10 може працювати на всіх сучасних гаджетах (смартфон, плашет, комп'ютер, ноутбук, телевізор). Особливістю Windows 10 є те, що вона може самостійно завантажити драйвер для незнайомого пристрою та підключити його до ОС.

Апаратні потреби для Windows 10, мінімальні:

– пам'ять: IA-32 1 Гб, x64 2 Гб;

– процесор: IA-32, x64, тактова частота 1 ГГц;

– вільний дисковий простір: IA-32 16 Гб, x64 20 Гб.

Windows 10 має наступні версії:

• Windows 10 Home – базова версія для користувачів ПК, лептопів і планшетів;

• Windows 10 Pro – версія для ПК, лептопів і планшетів з функціями для малого бізнесу;

• Windows 10 Mobile – версія для смартфонів і невеликих планшетів;

• Windows 10 Enterprise – версія для великого бізнесу з розширеними функціями керування корпоративними ресурсами, безпеки і т.д;

• Windows 10 Educational – варіант для навчальних закладів;

• Windows 10 Mobile Enterprise – варіант корпоративної версії, адаптованої під мобільні пристрої і з посиленою безпекою;

• Windows 10 IoT Core – версія для різноманітних комп'ютерних пристроїв, таких як термінали, роботи і т.д. із специфічними функціями, наприклад, для використання в платіжних терміналах на базі Windows, планшетів.

Операційна система Windows CE. Ця ОС відрізняється від інших хоча б тому, що вона призначена винятково для встановлення на кишенькові комп'ютери (palm-top). Такі мінікомп'ютери, що з'явилися наприкінці 90-х років, усього за кілька років зуміли набути поширення. Сьогодні «електронними органайзерами» користуються і ділові люди, які постійно перебувають у роз'їздах, і студенти.

У невеликій ОС інтегровані всі необхідні програми для роботи з мінікомп'ютером – простий текстовий редактор, записна книжка, електронна таблиця і система електронної пошти. Власники ПК навряд чи зіткнуться з цією ОС, а от власники різноманітних побутових пристроїв – цілком можливо. За задумом Microsoft, Windows CE незабаром буде встановлюватися навіть на бортові комп'ютери деяких моделей автомобілів. На даний час на ринку наладонних

комп'ютерів Windows CE не є лідером, поступаючись PalmOS та іншим конкуруючим продуктам.

Огляд деяких комерційних операційних систем реального часу. Основними відмінностями операційних систем реального часу (ОСРЧ) від ОС загального призначення є:

– орієнтація на оброблення зовнішніх подій;

– детермінований час реакції на зовнішню подію;

– модульна організація;

– невеликий розмір системи.

Операційна система OS-9. Операційна система OS-9 фірми Microware належить до класу UNIX-подібних ОСРЧ. По суті OS-9 є багатозадачною ОС із пріоритетною витісняльною диспетчеризацією. Ця ОС допускає можливість багатокористувацької роботи. Об'єктно-орієнтовний модульний дизайн системи дозволяє конфігурувати систему в дуже широкому діапазоні від вбудованих систем, до великих мережевих застосунків. Відповідно до цієї концепції всі функціональні компоненти OS-9 (ядро, ієрархічні файлові менеджери, драйвери пристроїв і т.ін.), реалізовані у вигляді незалежних модулів. Усі модулі ОС позиційно-незалежні й можуть бути розміщені в ПЗП, а також вилучатися із системи в процесі її функціонування без якої-небудь повторної інсталяції або перекомпонування. Ядро забезпечує основний системний сервіс, включаючи керування процесами й розподіл ресурсів.

Основні характеристики:

1. Архітектура: на основі мікроядра.

2. Стандарт: власний, виклики схожі на UNIX.

Властивості як ОСРЧ:

– багатозадачність: багатопроцесність;

– багатопроцесорність;

– рівні пріоритетів: 65535;

– час реакції: 3 мкс;

– планування: пріоритетне, FIFO, спеціальний механізм планування; витісняюче (preemptive) ядро.

3. Операційна система розроблення (host): UNIX/Windows.

4. Процесори (target): Motorola 68xxx, Intel 80×86, ARM, MIPS, PowerPC.

5. Лінії зв'язку (host-target): послідовний канал і ethernet.

6. Мінімальний розмір: 16 кбайт.

7. Засоби синхронізації й взаємодії: розподілювана пам'ять, сигнали, семафори, події.

Операційна система VxWorks. VxWorks належить до ОС «твердого» реального часу. Характерною особливістю цієї ОС є те, що завдяки її розвиненим мережевим можливостям усі розроблення ПЗ виконуються на інструментальному комп'ютері (хост-системі) з використанням крос-засобів для наступного виконання на цільовій машині під керуванням VxWorks.

Відмітна ознака системи – можливість керувати роботою складних комплексів реального часу й бортових пристроїв, що використовують процесорні елементи різних постачальників. Три основні компоненти цієї ОСРЧ утворюють єдине інтегроване середовище: власне ядро системи, що керує процесором; набір засобів міжпроцесорної взаємодії; комплект комунікаційних програм для роботи з Ethernet або послідовними каналами зв'язку.

Основні характеристики:

1. Архітектура: монолітна.

2. Стандарт: власний і POSIX 1003.

3. Властивості як ОСРЧ:

− багатозадачність: багатопроцесність;

− багатопроцесорність;

− рівні пріоритетів: 256;

− час реакції: 4 мкс;

− час перемикання контексту: 15 мкс;

− планування: пріоритетне; витісняюче (preemptive) ядро.

4. Операційна система розроблення (host): UNIX/Windows.

5. Процесори (target): Motorola 68xxx, Intel 80×86, Intel 80960, PowerPC, SPARC, Alpha, MIPS, ARM.

6. Лінії зв'язку host-target: послідовний канал, ethernet, шина VME.

7. Мінімальний розмір: 22 кбайт.

8. Засоби синхронізації й взаємодії: семафори POSIX 1003, черги, сигнали.

Операційна система QNX. Операційну систему QNX канадської компанії QNX Software System Ltd. побудовано на основі ієрархічної мікроядерної архітектури.

Мікроядро QNX виконує такі функції:

– міжпроцесорний обмін;

– низькорівневий мережевий обмін;

– диспетчеризація завдань;

– низкорівневе оброблення переривань.

Основні характеристики:

1. Архітектура: на основі мікроядра.

2. Стандарт: POSIX 1003.

3. Властивості як ОСРЧ:

– багатозадачність: POSIX 1003 (багатопроцесність і багатозадачність);

– багатопроцесорність;

– рівні пріоритетів: 32;

– час реакції: 4,3 мкс;

– час перемикання контексту: 13 мкс;

– планування: FIFO, round robin, адаптивне; витісняюче (preemptive) ядро.

4. Процесори (target): Intel 80×86.

5. Мінімальний розмір: 60 кбайт.

6. Засоби синхронізації й взаємодії: POSIX 1003 (семафори, mutex, condvar).

Операційна система LynxOS. Система LynxOS випускається фірмою Lynx Real Time Systems (Los Gatos, USA). ОСРЧ із клону UNIX-систем, що забезпечує детермінований час відгуку за запитами.

Основні характеристики:

1. Архітектура: на основі мікроядра.

2. Стандарт: POSIX 1003.

3. Властивості як ОСРЧ:

– багатозадачність POSIX 1003 (багатопроцесність та багатозадачність);

– багатопроцесорність;

– рівні пріоритетів: 255;

– час реакції: 7 мкс;

– час перемикання контексту: 17 мкс;

– планування: FIFO, round robin, Quantum, витісняюче (preemptive) ядро.

4. Процесори (target): Intel 80×86, Motorola 68xxx, SPARC, PowerPC.

5. Мінімальний розмір:

– повної системи: 256 кбайт.

– зрізаної системи: 124 кбайт.

– тільки ядра: 33 кбайт.

Систему можна записати в ROM.

6. Засоби синхронізації і взаємодії: POSIX 1003 (семафори, mutex, condvar).

Операційна система pSOS. Система pSOS випускається Integrated Systems (Santa Clara, USA).

Основні характеристики:

1. Архітектура: на основі мікроядра.

2. Стандарт: власний.

3. Властивості як ОСРЧ:
– багатозадачність: багатопроцесність;
– багатопроцесорність;
– рівні пріоритетів: 255;
– час реакції: 4 мкс;
– час перемикання контексту: 12мкс;
– планування: пріоритетне; витісняюче (preemptive) ядро.
4. Операційна система розроблення (host): UNIX/Windows.
5. Процесори (target): Motorola 68xxx, Intel 80×86, Intel 80960, ARM, MIPS, PowerPC.
6. Мінімальний розмір: 15 кбайт.
7. Засоби синхронізації й взаємодії: семафори, mutex, події і т.ін.

3 Архітектура та структура ОС

3.1. Архітектура ОС

Монолітна система. Структуру системи показано на рис. 2.5.

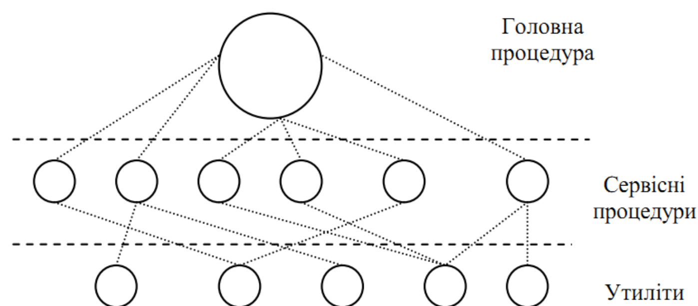

Рис. 2.5. Проста модель монолітної системи

Монолітна система має такі рівні:
1) головна процедура, яка викликає необхідні сервісні процедури;
2) набір сервісних процедур, що реалізують системні виклики;
3) набір утиліт, які обслуговують сервісні процедури.

У цій моделі для кожного системного виклику є одна сервісна процедура (наприклад, читати з файлу). Утиліти виконують функції, які потрібні декільком сервісним процедурам (наприклад, для зчитування й записування файлу необхідна утиліта роботи з диском).

Етапи оброблення виклику:
– приймається виклик;
– виконується перехід з режиму користувача в режим ядра;
– параметри виклику перевіряються ОС для визначення системного виклику;
− після цього ОС звертається до таблиці, яка містить посилання на процедури, та викликає відповідну процедуру.

Багаторівнева структура операційної системи. Узагальненням монолітної системи є організація ОС як ієрархії рівнів (рис. 2.6). Рівні утворюються групами функцій ОС, такими, як файлова система, керування процесами та пристроями і т.ін. Кожний рівень може взаємодіяти тільки з безпосереднім сусіднім рівнем – вищим або нижчим. Прикладні програми або модулі самої ОС передають запити вгору і вниз за цими рівнями.

<table>
<tr><th>Рівні</th><th colspan="3">Функції</th></tr>
<tr><td>7</td><td colspan="3">Оброблювач системних викликів</td></tr>
<tr><td>6</td><td>Файлова система 1</td><td></td><td>Файлова система n</td></tr>
<tr><td>5</td><td colspan="3">Віртуальна пам’ять</td></tr>
</table>

4	Драйвер 1	Драйвер 2	…	…	Драйвер *n*
3	Керування потоками				
2	Оброблення переривань, керування пам’яттю				
1	Приховування апаратури низького рівня				

Рис. 2.6. Приклад структури багаторівневої системи

Переваги: висока продуктивність.
Недоліки:
– великий код ядра і, як наслідок, великий вміст помилок;
– ненадійний захист ядра від допоміжних процесів.
Приклад реалізації багаторівневої моделі UNIX показано на рис. 2.7 і 2.8.

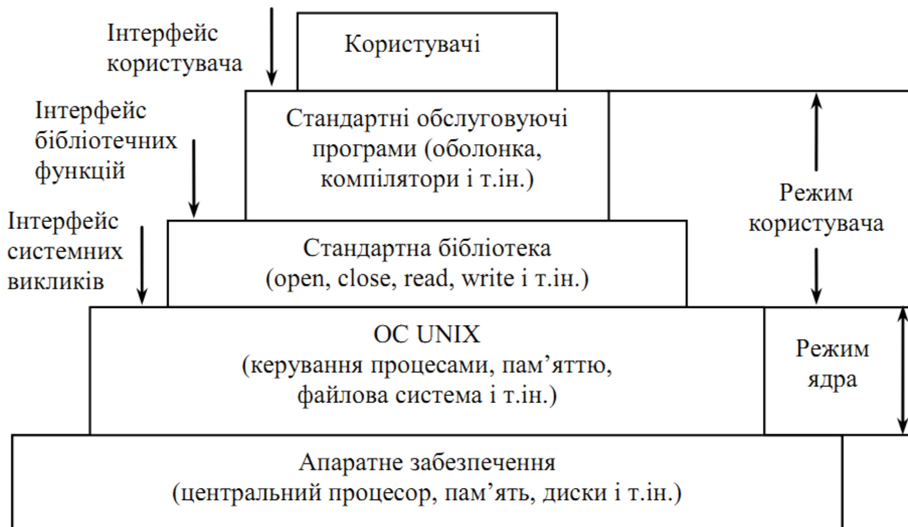

Рис. 2.7. Структура ОС UNIX

<table>
<tr><td colspan="5">Системні виклики</td><td colspan="3">Апаратні та емульовані переривання</td></tr>
<tr><td colspan="2">Керування терміналом</td><td>Сокети</td><td>Найме-нування файла</td><td>Відобра-ження адрес</td><td>Сторінкові перери-вання</td><td rowspan="2">Оброблення сигналів</td><td rowspan="2">Створення та завершення процесів</td></tr>
<tr><td rowspan="2">Необроблений телетайп</td><td>Оброблений телетайп</td><td>Мережеві протоколи</td><td>Файлові системи</td><td colspan="2">Віртуальна пам’ять</td></tr>
<tr><td>Дисципліни лінії зв’язку</td><td>Марш-рутизація</td><td>Буферний кеш</td><td colspan="2">Драйвери мережевих пристроїв</td><td colspan="2">Планування процесів</td></tr>
<tr><td colspan="2">Символьні пристрої</td><td>Драйвери мережевих пристроїв</td><td colspan="3">Драйвери дискових пристроїв</td><td colspan="2">Диспетчери-зація процесів</td></tr>
<tr><td colspan="8">Апаратура</td></tr>
</table>

Рис. 2.8. Ядро ОС UNIX

Приклад реалізації багаторівневої моделі Windows показано на рис. 2.9.

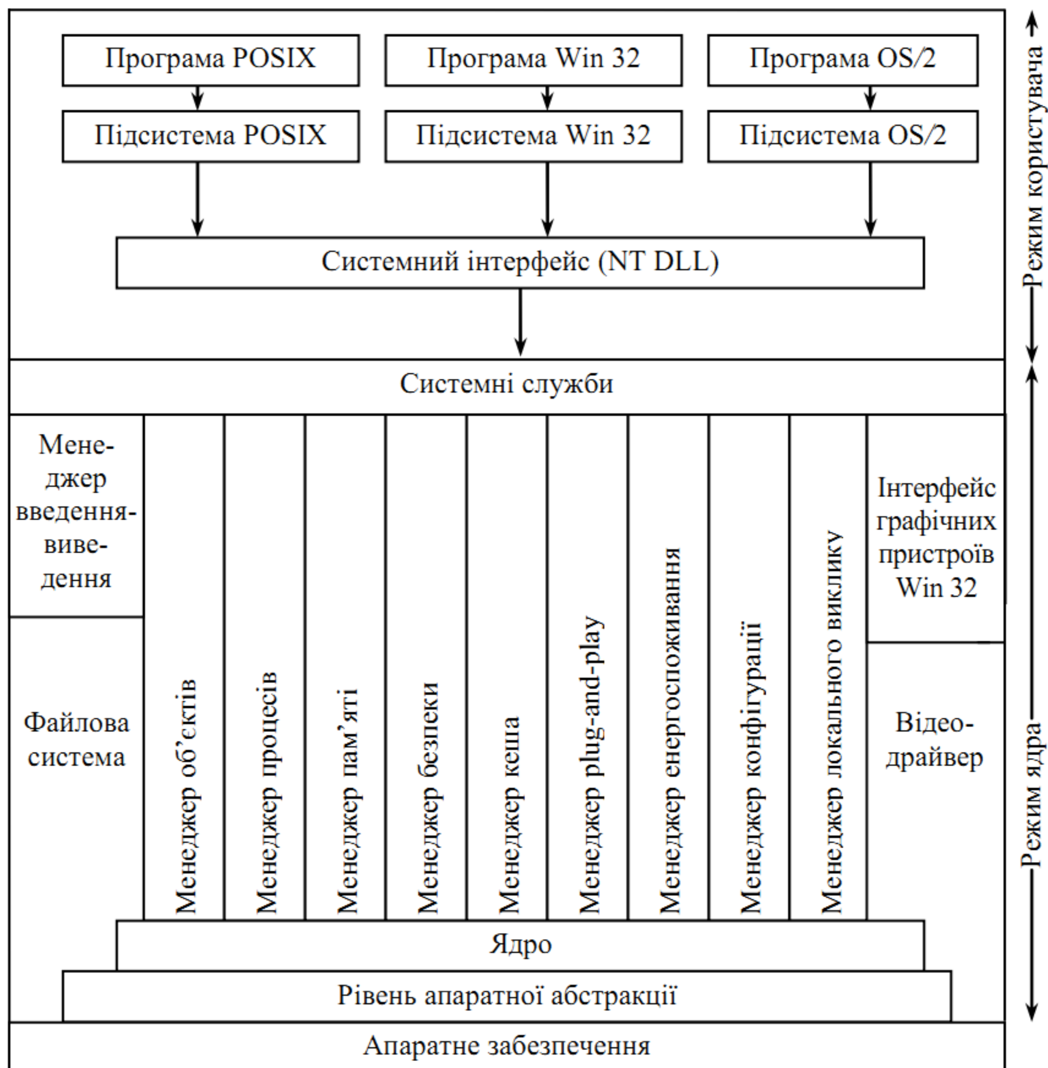

Рис. 2.9. Структура ОС Windows 2000

3.2 Мікроядерна архітектура

Концепція. Мікроядерна архітектура є альтернативою класичному способу побудови ОС. Під класичною архітектурою в цьому випадку розуміють структурну організацію ОС, відповідно до якої основні функції ОС, що складають багатошарове ядро, виконуються в привілейованому режимі. При цьому деякі допоміжні функції ОС оформлюються у вигляді застосунків і виконуються в користувацькому режимі поряд зі звичайними користувацькими програмами (стаючи системними утилітами або обробними програмами). Кожний застосунок користувацького режиму працює у власному адресному просторі й захищений тим самим від втручання інших застосунків. Код ядра, виконуваний у привілейованому режимі, має доступ до ділянок пам'яті всіх застосунків, але сам повністю від них захищений. Застосунки звертаються до ядра із запитами на виконання системних функцій.

Суть мікроядерної архітектури полягає в наступному. У привілейованому режимі залишається працювати тільки дуже невелика частина ОС, названа мікроядром (рис. 2.10). Мікроядро захищене від інших частин ОС і застосунків. До складу мікроядра входять машинозалежні модулі, а також модулі, що виконують базові функції ядра з керування процесами, оброблення переривань, керування віртуальною пам'яттю, пересилання

повідомлень і керування пристроями введення-виведення, пов’язані із завантаженням або зчитуванням регістрів пристроїв. Набір функцій мікроядра звичайно відповідає функціям шару базових механізмів звичайного ядра. Такі функції ОС важко, або навіть неможливо, виконати в просторі користувача.

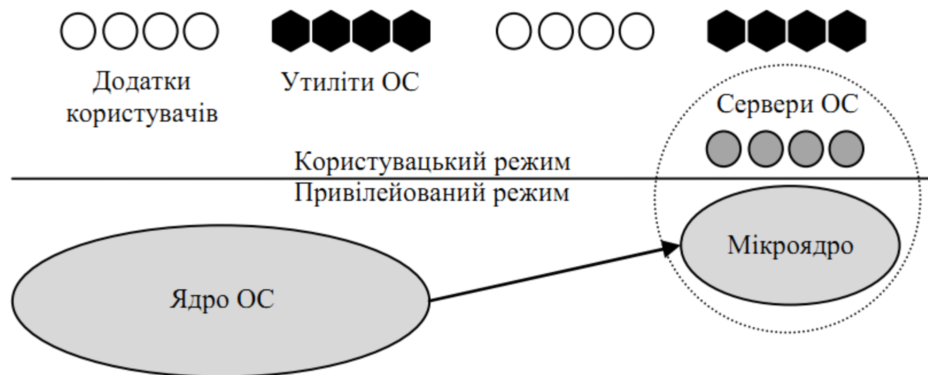

Рис. 2.10. Перенесення основного обсягу функцій ядра в користувацький простір

Усі інші більш високорівневі функції ядра оформлюються у вигляді застосунків, які працюють у користувацькому режимі. Однозначного рішення про те, які із системних функцій потрібно залишити в привілейованому режимі, а які перенести в користувацький, немає. У загальному випадку багато менеджерів ресурсів, що є невід’ємними частинами звичайного ядра – файлова система, підсистеми керування віртуальною пам’яттю й процесами, менеджер безпеки – стають «периферійними» модулями, що працюють у користувацькому режимі.

Працюючі в користувацькому режимі менеджери ресурсів мають принципові відмінності від традиційних утиліт і обробних програм ОС, хоча за мікроядерної архітектури всі ці програмні компоненти також оформлені у вигляді застосунків. Утиліти й обробні програми викликаються загалом користувачами. Ситуації, коли один застосунок потребує виконання функції (процедури) іншого застосунка, виникають украй рідко. Тому в ОС із класичною архітектурою немає механізму, за допомогою якого один застосунок міг би викликати функції іншого. Коли ж у формі застосунка оформлюється частина ОС, то за визначенням основним призначенням такого застосунка є обслуговування запитів інших застосунків, наприклад створення процесу, виділення пам’яті, перевірка прав доступу до ресурсу і т. ін. Саме тому менеджери ресурсів, винесені в користувацький режим, називаються серверами ОС, тобто модулями, основним призначенням яких є обслуговування запитів локальних застосунків та інших модулів ОС. Очевидно, що для реалізації мікроядерної архітектури необхідною умовою є наявність в ОС зручного й ефективного способу виклику процедур одного процесу з іншого. Підтримка такого механізму і є однією з головних функцій мікроядра. Схематично механізм звертання до функцій ОС, що оформлені у вигляді серверів, показано на рис. 2.11.

Клієнт, яким може бути або прикладна програма, або інший компонент ОС, запитує виконання якоїсь функції у відповідного сервера, посилаючи йому повідомлення. Безпосереднє передавання повідомлень між застосунками неможливе, оскільки їхні адресні простори ізольовані один від одного. Мікроядро, що виконується в привілейованому режимі, має доступ до адресних просторів кожного з цих застосунків і тому може працювати як посередник. Мікроядро спочатку передає повідомлення, яке містить ім’я й параметри викликуваної процедури, потрібне серверу; потім сервер виконує запитану операцію, після чого ядро повертає результати клієнту за допомогою іншого повідомлення. Таким чином, робота мікроядерної ОС відповідає відомій моделі клієнт-сервер, у якій роль транспортних засобів виконує мікроядро.

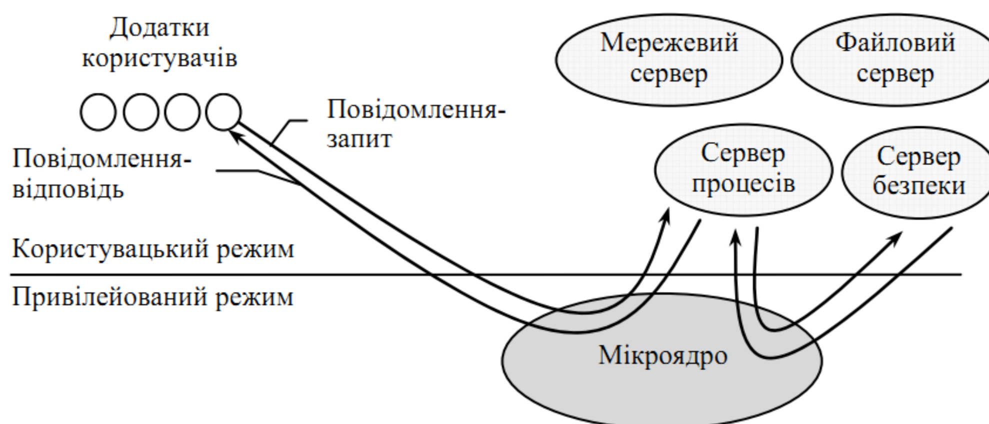

Рис. 2.11. Реалізація системного виклику в мікроядерній архітектурі

Переваги й недоліки мікроядерної архітектури. Операційні системи, засновані на концепції мікроядра, великою мірою задовольняють більшість вимог, поставлених до сучасних ОС, маючи переносимість, розширюваність, надійність і створюючи належні передумови для підтримання розподілених застосунків. За ці переваги доводиться поступатися зниженням продуктивності, і це є основним недоліком мікроядерної архітектури.

Високий ступінь переносимості зумовлюється тим, що весь машинозалежний код ізольований у мікроядрі, тому для перенесення системи на новий процесор потрібно менше змін, і всі вони логічно згруповані.

Надвисока розширюваність властива мікроядерній ОС. У традиційних системах навіть за наявності багатошарової структури нелегко вилучити один шар і поміняти його на інший через множинність і розмитість інтерфейсів між шарами. Додавання нових функцій і зміна існуючих потребують відповідного знання ОС та великих витрат часу. Водночас обмежений набір чітко визначених інтерфейсів мікроядра дає змогу впорядковано збільшуватись і розвиватися ОС. Додавання нової підсистеми потребує розроблення нового застосунка, що ніяк не стосується цілісності мікроядра. Мікроядерна структура дозволяє не тільки додавати, але й скорочувати кількість компонентів ОС, що також буває дуже корисно. Наприклад, не всім користувачам потрібні засоби безпеки або підтримки розподілених обчислень, а вилучати їх із традиційного ядра найчастіше неможливо. Звичайно традиційні ОС дозволяють динамічно додавати в ядро або вилучати з ядра тільки драйвери зовнішніх пристроїв – через часті зміни в конфігурації підключених до комп’ютера зовнішніх пристроїв підсистема введення-виведення ядра допускає завантаження й вивантаження драйверів «на ходу», але для цього вона розробляється особливим способом (наприклад, середовище STREAMS в UNIX або менеджер введення-виведення у Windows NT). За мікроядерного підходу конфігурованість ОС не створює проблем і не потребує особливих заходів – достатньо змінити файл із налагодженнями початкової конфігурації системи або ж зупинити не потрібні більше сервери в ході роботи звичайними для зупинення застосунків засобами.

Використання мікроядерної моделі підвищує надійність ОС. Кожний сервер виконується як окремий процес у власній ділянці пам’яті й у такий спосіб захищений від інших серверів ОС, що не спостерігається в традиційній ОС, де всі модулі ядра можуть впливати один на одного. І якщо окремий сервер пошкоджується, то він може бути перезапущений без зупинення або пошкодження інших серверів ОС. Більше того, оскільки сервери виконуються в користувацькому режимі, вони не мають безпосереднього доступу до апаратури й не можуть модифікувати пам’ять, у якій зберігається й працює мікроядро. Іншим потенційним джерелом підвищення надійності ОС є зменшений об’єм коду мікроядра порівняно з традиційним ядром – це знижує ймовірність появи помилок програмування.

Модель із мікроядром добре підходить для підтримання розподілених обчислень, оскільки в ній використовуються механізми, аналогічні мережевим: взаємодія клієнтів і серверів через

обмін повідомленнями. Сервери мікроядерної ОС можуть працювати як на одному, так і на різних комп'ютерах. Отримавши повідомлення від застосунка, мікроядро може обробити його самостійно й передати локальному серверу або ж переслати по мережі мікроядру, що працює на іншому комп'ютері. Перехід до розподіленого оброблення потребує мінімальних змін у роботі ОС – локальний транспорт заміняється на мережевий.

Продуктивність. За класичної організації ОС (рис. 2.12, а) виконання системного виклику супроводжується двома перемиканнями режимів, а в разі мікроядерної організації (рис. 2.12, б) – чотирма. Таким чином, ОС на основі мікроядра за інших рівних умов завжди буде менш продуктивною, ніж ОС із класичним ядром. Саме з цієї причини мікроядерний підхід не дістав очікуваного поширення.

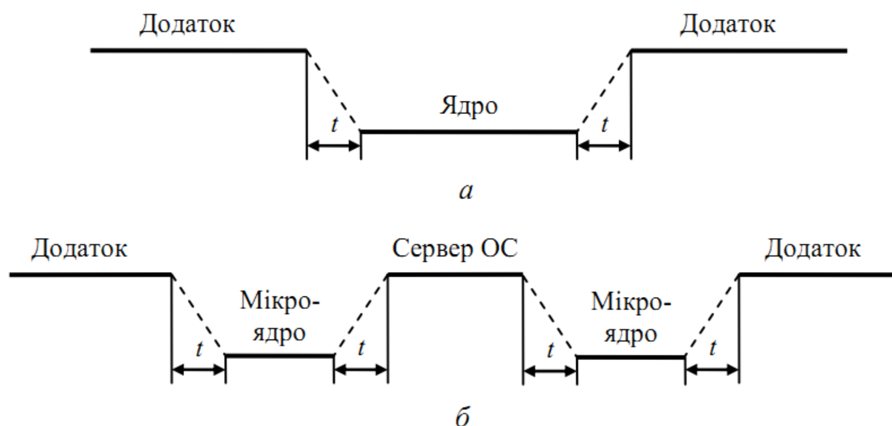

Рис. 2.12. Зміна режимів для виконання системного виклику

Недолік мікроядерного підходу добре ілюструє історія розвитку Windows NT. У версіях 3.1 і 3.5 диспетчер вікон, графічна бібліотека й високорівневі драйвери графічних пристроїв входили до складу серверу користувацького режиму, і виклик функцій цих модулів здійснювався відповідно до мікроядерної схеми. Однак незабаром розробники Windows NT зрозуміли, що такий механізм звертань до часто використовуваних функцій графічного інтерфейсу істотно сповільнює роботу застосунків і робить ОС вразливою в умовах гострої конкуренції. В результаті версію Windows NT 4.0 істотно змінено — всі наведені вище модулі перенесено в ядро, що віддалило цю ОС від ідеальної мікроядерної архітектури, та зате різко підвищило її продуктивність.

Цей приклад ілюструє головну проблему, з якою стикаються розробники ОС, які вирішили застосувати мікроядерний підхід, – що включати в мікроядро, а що виносити в користувацький простір. В ідеальному випадку мікроядро може складатися тільки із засобів передавання повідомлень, засобів взаємодії з апаратурою, зокрема засобів доступу до механізмів привілейованого захисту. Однак багато розробників не завжди жорстко дотримуються принципу мінімізації функцій ядра, часто жертвуючи цим заради підвищення продуктивності. В результаті реалізації ОС утворюють деякий спектр, на одному краю якого розміщені системи з мінімально можливим мікроядром, а на іншому — системи, подібні до Windows NT, у яких мікроядро виконує досить великий обсяг функцій.

3.3. Екзоядро

Екзоядро – один з сучасних напрямків подальшого розвитку мікроядерної архітектури. Це ядро ОС, що надає лише функції для взаємодії між процесами і безпечного виділення і звільнення ресурсів. Надання прикладним програмам абстракцій для фізичних ресурсів не входить в обов'язки екзоядра. Ці функції виносяться в бібліотеку захищеного режиму — так звану libOS, яка може забезпечувати довільний набір абстракцій, сумісний з тією або іншою вже існуючою ОС, наприклад Linux або Windows.

В порівнянні з ОС на основі мікроядер, екзоядра забезпечують набагато більшу ефективність за рахунок відсутності перемикання процесів при кожному зверненні до апаратного устаткування.

3.4. Наноядро

Наноядро – архітектура ядра ОС, в рамках якої вкрай спрощене ядро виконує лише одне завдання – обробку апаратних переривань, що генеруються пристроями комп'ютера. Після обробки переривань від апаратури наноядро, у свою чергу, посилає інформацію про результати обробки вищерозміщеному програмному забезпеченню за допомогою того ж механізму переривань.

Найчастіше в сучасних обчислювальних системах наноядра використовуються для **віртуалізації** апаратного забезпечення з метою дозволити кільком різним ОС працювати одночасно і паралельно на одному і тому ж комп'ютері. Наноядра також використовуються для забезпечення переносимості ОС на різне апаратне забезпечення або для забезпечення можливості запуску ОС на новому, несумісному апаратному забезпеченні без її повного переписування і перекомпіляції.

Найбільш відомі приклади використання наноядер – сервер віртуальних машин Vmware ESX Server, ядро для Mac OS Classic з процесором POWERPC, яке емулювало для ОС апаратуру процесорів Motorola 680x0, ядро Adeos, що працює як модуль ядра для Linux і дозволяє виконувати одночасно з Linux яку-небудь іншу ОС.

4. Ядро та багатошарова структура ОС

4.1. Ядро та допоміжні модулі ОС

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

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

Модулі ядра виконують такі базові функції ОС, як керування процесами, пам'яттю, пристроями введення-виведення і т.ін. Ядро є серцевиною ОС, без нього ОС є повністю непрацездатною і не зможе виконати жодної зі своїх функцій.

До складу ядра входять функції, спрямовані на виконання внутрішньосистемних завдань з організації обчислювального процесу, а саме: перемикання контекстів, завантаження-вивантаження сторінок, оброблення переривань. Ці функції недоступні для застосунків. Інший клас функцій ядра служить для підтримки застосунків, створюючи для них прикладне програмне середовище. Застосунки можуть звертатися до ядра із запитами – системними викликами – для виконання тих або інших дій, наприклад для відкриття і зчитування файлу, виведення графічної інформації на дисплей, отримання системного часу і т.ін. Функції ядра, які можуть викликатися застосунками, утворюють інтерфейс прикладного програмування – API.

Функції, що виконуються модулями ядра, є найбільш часто використовуваними функціями ОС, тому швидкість їх виконання визначає продуктивність системи в цілому. Для забезпечення високої швидкості роботи ОС усі модулі ядра або більша їх частина постійно перебувають в оперативній пам'яті, тобто є *резидентними*.

Ядро є рушійною силою всіх обчислювальних процесів у комп'ютерній системі, і руйнування ядра рівносильне руйнуванню всієї системи. Тому розробники ОС приділяють особливу увагу надійності кодів ядра, у результаті процес їх налагодження може тривати багато місяців.

Ядру звичайно надають вигляду програмного модуля деякого спеціального формату, що відрізняється від формату користувацьких застосунків.

Інші модулі ОС виконують дуже корисні, але менш обов'язкові функції. Наприклад, такими допоміжними модулями можуть бути програми архівації даних, дефрагментації диска,

текстові редактори. Допоміжні модулі ОС оформляють у вигляді або застосунків, або бібліотек процедур.

Оскільки деякі компоненти ОС мають вигляд звичайних програм, тобто вигляд виконуваних модулів стандартного для цієї ОС формату, то часто дуже складно чітко розмежувати ОС і застосунки.

Рішення про те, чи є яка-небудь програма частиною ОС чи не є нею, приймає виробник ОС. Серед багатьох факторів, здатних вплинути на це рішення, важливими є перспективи того, чи буде програма мати масовий попит у потенційних користувачів цієї ОС.

Деяка програма може існувати певний час як користувацький застосунок, а потім стати частиною ОС, або навпаки. Яскравим прикладом такої зміни статусу програми є Web-переглядач компанії Microsoft, який спочатку поставлявся як окремий застосунок, а потім став частиною ОС Windows.

Допоміжні модулі ОС зазвичай поділяють на такі групи:

− *утиліти*, це програми, які вирішують окремі завдання керування і супроводу комп'ютерної системи (наприклад, програми стиску дисків, архівації даних);

− *системні обробні програми*, це текстові або графічні редактори, компілятори, компонувачі, налагоджувачі;

− *програми надання користувачу додаткових послуг* – спеціальний варіант інтерфейсу користувача, калькулятор і навіть ігри;

− **бібліотеки процедур різного призначення**, що спрощують розроблення застосунків (наприклад, бібліотека математичних функцій, функцій введення-виведення).

Як і звичайні програми, для виконання своїх завдань утиліти, системні обробні програми та бібліотеки ОС, звертаються до функцій ядра за допомогою системних викликів (рис. 2.13).

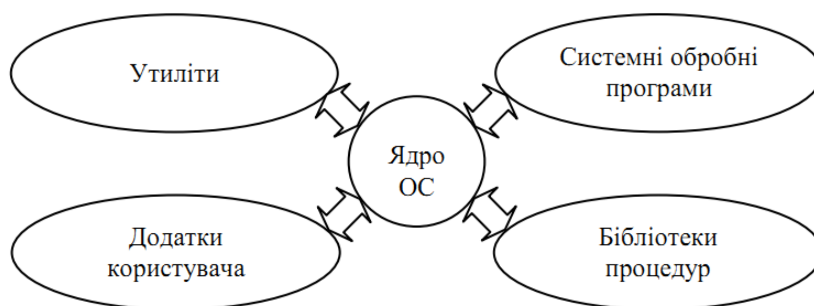

Рис. 2.13. Взаємодія між ядром і допоміжними модулями ОС

Поділ ОС на ядро і модулі-застосунки забезпечує легку розширюваність ОС. Щоб додати нову високорівневу функцію, достатньо розробити новий застосунок, і при цьому не потрібно модифіковувати важливі функції, що утворюють ядро системи. Однак внесення змін у функції ядра може виявитися набагато складнішим, і складність ця залежить від структурної організації самого ядра. У деяких випадках кожне виправлення ядра може вимагати його повної перекомпіляції. Модулі ОС, оформлені у вигляді утиліт, системних обробних програм і бібліотек, звичайно завантажуються в опертивну пам'ять лише на час виконання своїх функцій, тобто є *транзитними*. Постійно в оперативній пам'яті розміщуються тільки найнеобхідніші коди ОС, які становлять її ядро, що економить оперативну пам'ять комп'ютера.

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

4.2 Ядро в привілейованому режимі

Для надійного керування ходом виконання застосунків ОС повинна мати стосовно застосунків певні привілеї. Інакше програма, що некоректно працює, може втрутитися в роботу ОС і, наприклад, зруйнувати частину її кодів. Усі зусилля розробників ОС виявляться марними,

якщо їх рішення втілені в незахищені від застосунків модулі системи. Операційна система повинна мати виняткові повноваження для того, щоб відігравати роль арбітра в суперечці застосунків за ресурси комп'ютера в мультипрограмному режимі. Жодна програма не повинна мати можливості без відома ОС отримувати додаткову ділянку пам'яті, займати процесор довше дозволеного ОС періоду часу, безпосередньо керувати спільно використовуваними зовнішніми пристроями.

Забезпечити привілеї ОС неможливо без спеціальних засобів апаратної підтримки. Апаратура комп'ютера повинна підтримувати як мінімум два режими роботи – режим користувача (user mode) і привілейований режим, який також називають режимом ядра (kernel mode), або режимом супервізора (supervisor mode). Мається на увазі, що ОС або деякі її частини працюють в привілейованому режимі, а застосунки – у режимі користувача.

Оскільки ядро виконує всі основні функції ОС, то найчастіше саме ядро стає тією частиною ОС, що працює в привілейованому режимі (рис. 2.14). Іноді ця властивість – робота в привілейованому режимі – є основним визначенням поняття «ядро».

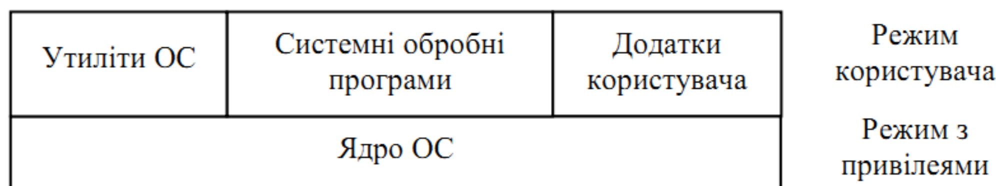

Рис. 2.14. Архітектура ОС з ядром у привілейованому режимі

Програми залежать від заборони виконання в режимі користувача деяких критичних команд, пов'язаних з перемиканням процесора із завдання на завдання, керуванням пристроями введення-виведення, доступом до механізмів розподілу та захисту пам'яті. Виконання деяких інструкцій в режимі користувача забороняється безумовно (очевидно, що до таких інструкцій належить інструкція переходу в привілейований режим), тоді як інші інструкції забороняється виконувати тільки за певних умов. Наприклад, інструкції введення-виведення можуть бути заборонені програмам при доступі до контролера жорсткого диска, який зберігає дані, загальні для ОС і всіх застосунків, але дозволені для доступу до послідовного порту, який виділений у монопольне володіння для певної програми. Важливо, що умови дозволу виконання критичних інструкцій перебувають під повним контролем ОС і цей контроль забезпечується за рахунок набору інструкцій, безумовно заборонених для режиму користувача.

Аналогічним чином забезпечуються привілеї при доступі до пам'яті. Наприклад, виконання інструкції доступу до пам'яті для програми дозволяється, якщо інструкція звертається до ділянки пам'яті, відведеної для застосунка ОС, і забороняється під час звернення до ділянок пам'яті, зайнятих ОС або іншими застосунками. Повний контроль ОС над доступом до пам'яті досягається за рахунок того, що інструкцію або інструкції конфігурування механізмів захисту пам'яті (наприклад, зміни ключів захисту пам'яті в мейнфреймах IBM або вказівник таблиці дескрипторів пам'яті в процесорах Pentium) дозволяється виконувати тільки в привілейованому режимі.

Дуже важливо, що механізми захисту пам'яті використовуються ОС не тільки для захисту власних ділянок пам'яті від застосунків, але і для захисту ділянок пам'яті, виділених ОС будь-якому застосунку, від інших застосунків. Кожен застосунок працює у власному адресному просторі. Ця властивість дозволяє локалізувати застосунок, що некоректно працює, у власній ділянці пам'яті, тому його помилки не впливають на інші програми і ОС.

Між кількістю рівнів привілеїв, що реалізуються апаратно, і кількістю рівнів привілеїв, підтримуваних ОС, немає прямої відповідності. Так, на базі чотирьох рівнів, що забезпечуються процесорами компанії Intel, ОС OS/2 будує трирівневу систему привілеїв, а ОС Windows NT, UNIX і деякі інші обмежуються дворівневою системою.

Однак, якщо апаратура підтримує хоча б два рівні привілеїв, то ОС може на цій основі створити програмним способом як завгодно розвинену систему захисту.

Ця система може, наприклад, підтримувати декілька рівнів привілеїв, що утворюють ієрархію. Наявність декількох рівнів привілеїв дозволяє більш тонко розподіляти повноваження як між модулями ОС, так і між самими програмами. Поява всередині ОС більш привілейованих і менш привілейованих частин дозволяє підвищити стійкість ОС до внутрішніх помилок програмних кодів, оскільки такі помилки будуть поширюватися тільки всередині модулів з певним рівнем привілеїв. Диференціація привілеїв у середовищі прикладних модулів дозволяє будувати складні прикладні комплекси, у яких частина більш привілейованих модулів може, отримувати доступ до даних менш привілейованих модулів і керувати їх виконанням.

На основі двох режимів привілеїв процесора ОС може побудувати складну систему індивідуального захисту ресурсів, прикладом якої є типова система захисту файлів і каталогів. Така система до зволяє задати для будь-якого користувача певні права доступу до кожного з файлів і каталогів.

Підвищення стійкості ОС, що забезпечується переходом ядра в привілейований режим, досягається деяким уповільненням виконання системних викликів. Системний виклик привілейованого ядра ініціює перемикання процесора з режиму користувача в привілейований, а в разі повернення до застосунка – перемикання з привілейованого режиму в режим користувача (рис. 2.15).

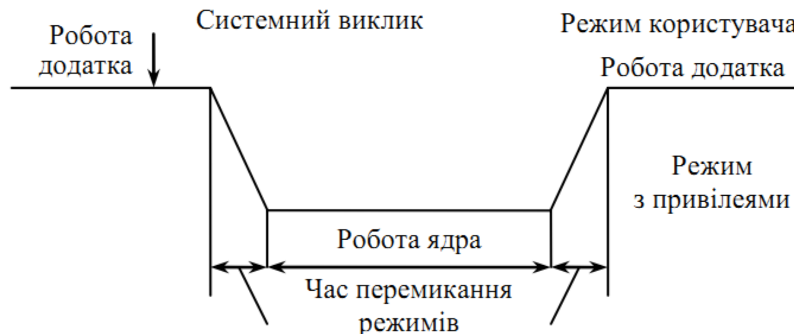

Рис. 2.15. Зміна режимів під час виконання системного виклику до привілейованого ядра

У всіх типах процесорів, через додаткове дворазове затримання перемикання, перехід на процедуру зі зміною режиму виконується повільніше, ніж виклик процедури без зміни режиму. Архітектура ОС, побудована на привілейованому ядрі та застосунках режиму користувача, стала, по суті, класичною. Її використовують багато популярних ОС, зокрема численні версії UNIX, VAX VMS, OS/2 і з певними модифікаціями – Windows NT та ін.

У деяких випадках розробники ОС відступають від цього класичного варіанта архітектури, організовуючи роботу ядра та програм в одному і тому ж режимі. За такої побудови ОС звернення застосунків до ядра виконуються швидше, оскільки немає перемикання режимів, однак немає надійного апаратного захисту пам’яті, займаної модулями ОС, від некоректно працюючого застосунка. Розробники спеціалізованої ОС NetWare компанії Novell вдалися до такого потенційного зниження надійності власної ОС, оскільки обмежений набір її спеціалізованих застосунків дозволяє компенсувати цей архітектурний недолік ретельним налагодженням кожної програми.

В одному режимі працюють також ядро і програми тих ОС, що розроблені для процесорів, які взагалі не підтримують привілейований режим роботи. Найпоширенішим процесором такого типу був процесор Intel 8088/86, що став основою для ПК компанії IBM. Операційна система MS-DOS, розроблена компанією Microsoft для цих комп’ютерів, складалася з двох модулів `msdos.sys` і `io.sys`, що становили ядро системи (хоча назва «ядро» для цих модулів не вживалася, за своїм змістом вони ним були), до яких із системними викликами зверталися командний інтерпретатор `command.com`, системні утиліти та застосунки. Архітектура MS-

DOS відповідає архітектурі ОС. Некоректно написані програми цілком могли зруйнувати основні модулі MS-DOS, що іноді й відбувалося, але до використання MS-DOS (і багатьох подібних їй ранніх ОС для ПК, таких як MSX, CP/M) і не ставилися високі вимоги до надійності ОС.

4.3. Багатошарова структура ОС

Обчислювальну систему, якою керує ОС на основі ядра, можна розглядати як систему, що складається з трьох ієрархічно розміщених шарів. На нижньому шарі міститься апаратура, на проміжному – ядро, а на верхньому – утиліти, системні обробні програми й застосунки (рис. 2.16). Шарову структуру обчислювальної системи зображають у вигляді системи концентричних кіл, ілюструючи той факт, що кожен шар може взаємодіяти тільки із суміжними шарами. Дійсно, за такої організації ОС програми не можуть безпосередньо взаємодіяти з апаратурою, а тільки через шар ядра.

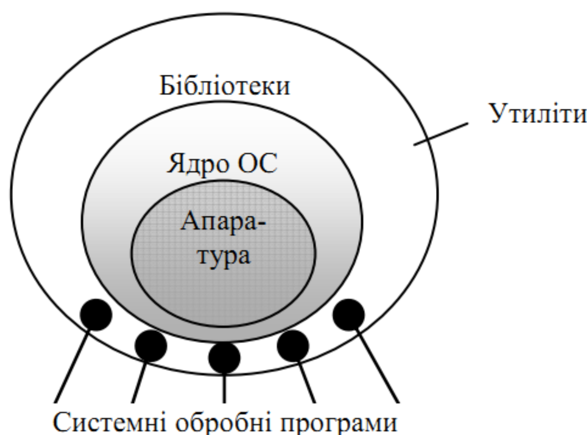

Рис. 2.16. Тришарова схема обчислювальної системи

Багатошаровий підхід є універсальним і ефективним способом декомпозиції складних систем будь-якого типу, у тому числі і програмних. Згідно з цим підходом система складається з ієрархії шарів. Кожен шар обслуговує розміщений вище шар, виконуючи для нього деякий набір функцій, які утворюють міжшаровий інтерфейс. На основі функцій розміщеного нижче шару наступний (вгору за ієрархією) шар будує свої функції – більш складні й потужніші, які, в свою чергу, виявляються примітивами для створення ще більш потужних функцій розміщеного вище шару. Жорсткі правила стосуються тільки взаємодії між шарами системи, а між модулями всередині шару зв'язки можуть бути довільними. Окремий модуль може виконати роботу або самостійно, або звернутися до іншого модуля свого шару, або звернутися за допомогою до нижчого рівня шару через міжшаровий інтерфейс.

Така організація системи має багато переваг. Вона суттєво спрощує розроблення системи, оскільки дозволяє спочатку визначити функції шарів і міжшарові інтерфейси «згори вниз», а потім за детальної реалізації поступово нарощувати потужність функцій шарів, рухаючись «знизу вгору». Крім того, у процесі модернізації системи можна змінювати модулі всередині шару без необхідності робити будь-які зміни в інших шарах, якщо ці внутрішні зміни не впливають на міжшарові інтерфейси. Оскільки ядро являє собою складний багатофункціональний комплекс, то багатошаровий підхід зазвичай поширюється і на структуру ядра.

Ядро може складатися з таких шарів: засобів апаратної підтримки, машинозалежних модулів, базових механізмів ядра, менеджерів ресурсів, інтерфейсу системних викликів.

Засоби апаратної підтримки ОС. Операційна система є комплексом програм, але частину функцій ОС можуть виконувати і апаратні засоби. Тому іноді ОС визначають як сукупність програмних і апаратних засобів (рис. 2.17). До ОС належать не всі апаратні пристрої

комп'ютера, а лише засоби апаратної підтримки ОС, тобто ті, які безпосередньо беруть участь в організації обчислювальних процесів: засоби підтримки привілейованого режиму, система переривань, засоби перемикання контекстів процесів, засоби захисту ділянок пам'яті і т.ін.

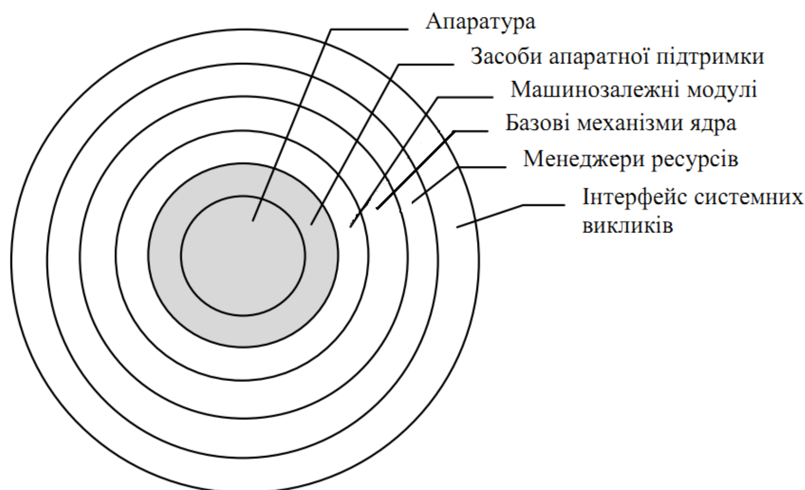

Рис. 2.17. Багатошарова структура ядра ОС

Машинозалежні модулі ОС. Цей шар утворюють програмні модулі, в яких відображається специфіка апаратної платформи комп'ютера. В ідеалі цей шар повністю екранує розміщені вище шари ядра від особливостей апаратури. Це дозволяє розробляти розміщені вище шари на основі машинонезалежних модулів, що існують у єдиному екземплярі для всіх типів апаратних платформ, які підтримує ця ОС. Як приклад екрануючого шару можна навести шар HAL ОС Windows NT.

Базові механізми ядра. Цей шар виконує найбільш примітивні операції ядра, такі як програмне перемикання контекстів процесів, диспетчеризацію переривань, переміщення сторінок з пам'яті на диск і назад і т. ін. Модулі цього шару не приймають рішень про розподіл ресурсів – вони тільки відпрацьовують прийняті «вгорі» рішення, що стало приводом називати їх виконавчими механізмами для модулів верхніх шарів. Наприклад, рішення про те, що в заданий момент часу потрібно перервати виконання поточного процесу *A* і почати виконання процесу *B*, приймається менеджером процесів на шарі, розміщеному вище, а шару базових механізмів передається тільки директива про потребу виконати перемикання з контексту поточного процесу на контекст процесу *B*.

Менеджери ресурсів. Цей шар складається з потужних функціональних модулів, що реалізують стратегічні завдання з керування основними ресурсами обчислювальної системи. Звичайно на цьому шарі працюють менеджери (диспетчери) процесів, введення-виведення, файлової системи й оперативної пам'яті. Розбиття на менеджери може бути і трохи іншим, наприклад менеджер файлової системи іноді поєднують з менеджером введення-виведення, а функції керування доступом користувачів до системи в цілому та до її окремих об'єктів доручають окремому менеджеру безпеки. Кожен з менеджерів веде облік вільних і використовуваних ресурсів певного типу і планує їх розподіл відповідно до запитів застосунків. Наприклад, менеджер віртуальної пам'яті керує переміщенням сторінок з оперативної пам'яті на диск і назад. Менеджер має відстежувати інтенсивність запитів сторінок, час перебування їх у пам'яті, стани процесів, що використовують дані, й багато інших параметрів, на підставі яких він час від часу приймає рішення про те, які сторінки необхідно вивантажити і які завантажити. Для виконання прийнятих рішень менеджер звертається до нижчого рівня шару базових механізмів із запитами про завантаження (вивантаження) конкретних сторінок. Усередині шару менеджерів існують тісні взаємні зв'язки, що відображають той факт, що для виконання процесу потрібен доступ одночасно до декількох ресурсів – процесора, ділянки пам'яті,

можливо, до певного файлу або пристрою введення-виведення. Наприклад, під час створення процесу менеджер процесів звертається до менеджера пам'яті, який повинен виділити процесу певну ділянку пам'яті для його кодів і даних.

Інтерфейс системних викликів. Цей шар є найбільш верхнім шаром ядра й взаємодіє безпосередньо із застосунками та системними утилітами, утворюючи прикладний програмний інтерфейс ОС (API). Функції API, що обслуговують системні виклики, надають доступ до ресурсів системи в зручній і компактній формі без вказівки деталей їх фізичного розміщення. Наприклад, в ОС UNIX за допомогою системного виклику

```
fd = open("/doc/a.txt", 0_RDONLY);
```

програма відкриває файл `a.txt`, що зберігається в каталозі `/doc`, а за допомогою системного виклику

```
read(fd,buffer,count);
```

зчитує з цього файла в ділянку власного адресного простору, що має ім'я buffer, деяку кількість байтів. Для здійснення таких комплексних дій системні виклики звичайно звертаються за допомогою до функцій шару менеджерів ресурсів, причому для виконання одного системного виклику може знадобитися декілька таких звернень.

Наведене розбиття ядра ОС на шари є досить умовним. У реальній системі кількість шарів і розподіл функцій між ними можуть бути й іншими. У системах, призначених для апаратних платформ одного типу, наприклад ОС NetWare, шар машинозалежних модулів звичайно не виділяється, зливаючись із шаром базових механізмів і, частково, з шаром менеджерів ресурсів. Не завжди оформляються в окремий шар базові механізми – в цьому випадку менеджери ресурсів не тільки планують використання ресурсів, а й самостійно реалізують свої плани.

Можлива й протилежна картина, коли ядро складається з більшої кількості шарів. Наприклад, менеджери ресурсів, складаючи певний шар ядра, в свою чергу, можуть мати багатошарову структуру. Перш за все це стосується менеджера введення-виведення, нижній шар якого складають драйвери пристроїв, наприклад драйвер жорсткого диска або драйвер мережевого адаптера, а верхні шари – драйвери файлових систем або протоколів мережевих служб, які відповідають за логічну організацію інформації.

Спосіб взаємодії шарів у реальній ОС також може відхилятися від описаної вище схеми. Для прискорення роботи ядра в деяких випадках відбувається безпосереднє звернення з верхнього шару до функцій нижніх шарів, оминаючи проміжні. Типовим прикладом такої «неправильної» взаємодії є початкова стадія оброблення системного виклику. На багатьох апаратних платформах для реалізації системного виклику використовується інструкція програмного переривання. Цим застосунок фактично викликає модуль первинного оброблення переривань, який міститься в шарі базових механізмів, а вже цей модуль викликає потрібну функцію з шару системних викликів. Самі функції системних викликів також іноді порушують субординацію ієрархічних шарів, звертаючись прямо до базових механізмів ядра.

Вибір кількості шарів ядра є відповідальним і складним завданням: збільшення кількості шарів веде до деякого уповільнення роботи ядра за рахунок додаткових накладних витрат на міжшарову взаємодію, а зменшення кількості шарів погіршує розширюваність і логічність системи. Звичайно ОС, що пройшли довгий шлях еволюційного розвитку, мають невпорядковане ядро з невеликою кількістю чітко виділених шарів, а у порівняно «молодих» ОС ядро розділене на більшу кількість шарів і їх взаємодія формалізована набагато більше.

5 Апаратна залежність і переносимість операційних систем

5.1 Апаратна залежність ОС

Багато ОС успішно працюють на різних апаратних платформах без істотних змін у своєму складі. Багато в чому це пояснюється тим, що, незважаючи на відмінності в деталях, засоби апаратної підтримки ОС більшості комп'ютерів набули тепер багато типових ознак, зокрема ці

засоби впливають передусім на роботу компонентів ОС. У результаті в ОС можна виділити досить компактний шар машинозалежних компонентів ядра й зробити інші шари ОС загальними для різних апаратних платформ.

5.2 Типові засоби апаратної підтримки ОС

Чіткої межі між програмною й апаратною реалізацією функцій ОС немає – рішення про те, які функції ОС будуть виконуватися програмно, а які апаратно, приймають розробники апаратного й програмного забезпечення комп'ютера. Проте всі сучасні апаратні платформи мають деякий типовий набір засобів апаратної підтримки ОС, зокрема таких:

– засобів підтримки привілейованого режиму;
– засобів трансляції адрес;
– засобів перемикання процесів;
– системи переривань;
– системного таймера;
– засобів захисту ділянок пам'яті.

Засоби підтримки привілейованого режиму звичайно ґрунтуються на системному регістрі процесора, який часто називають «словом стану» машини або процесора. Цей регістр має деякі ознаки, що обумовлюють режими роботи процесора, у тому числі й ознаку поточного режиму привілеїв. Зміна режиму привілеїв виконується за рахунок зміни «слова стану» машини внаслідок переривання або виконання привілейованої команди. Кількість градацій привілейованості може бути різною у різних типів процесорів, найбільш часто використовуються два рівні (ядро – користувач) або чотири (наприклад, ядро – супервізор – виконання – користувач у платформи VAX або 0–1–2–3 у процесорів Intel x86/Pentium). Засобами підтримки привілейованого режиму є перевірка допустимості виконання активною програмою інструкцій процесора за поточного рівня привілейованості.

Засоби трансляції адрес виконують операції перетворення віртуальних адрес, які містяться в кодах процесу, в адреси фізичної пам'яті. Таблиці, призначені для трансляції адрес, звичайно мають великий обсяг, тому для їх зберігання використовуються ділянки оперативної пам'яті, а апаратура процесора містить тільки вказівники на ці ділянки. Засоби трансляції адрес використовують дані вказівники для доступу до елементів таблиць і апаратного виконання алгоритму перетворення адреси, що значно прискорює процедуру трансляції порівняно з її чисто програмною реалізацією.

Засоби перемикання процесів призначені для швидкого збереження контексту процесу, що припиняється, й відновлення контексту процесу, який стає активним. Вміст контексту містить усі регістри загального призначення процесора, регістр прапорів операцій (тобто прапорів нуля, перенесення, переповнення і т.ін.), а також ті системні регістри і вказівники, які пов'язані з окремим процесом, а не з ОС, наприклад, вказівника на таблицю трансляції адрес процесу. Для зберігання контекстів припинених процесів використовуються ділянки оперативної пам'яті, які підтримуються вказівниками процесора.

Перемикання контексту виконується за певними командами процесора, наприклад за командою переходу до виконання нового завдання. Така команда викликає автоматичне завантаження даних зі збереженого контексту в регістри процесора, після чого процес триває з перерваного раніше місця.

Система переривань дозволяє комп'ютеру реагувати на зовнішні події, синхронізувати виконання процесів і роботу пристроїв введення-виведення, швидко переходити з однієї програми на іншу. Механізм переривань потрібен для того, щоб сповістити процесор про виникнення в обчислювальній системі якоїсь непередбаченої події або події, яка не синхронізована з циклом роботи процесора. Як приклади таких подій можна навести завершення операції введення-виведення зовнішнім пристроєм (наприклад, запис блоку даних контролером диска), некоректне завершення арифметичної операції (наприклад, переповнення регістра), закінчення інтервалу астрономічного часу. За наявності умов переривання його

джерело (контролер зовнішнього пристрою, таймер, арифметичний блок процесора тощо) виставляє певний електричний сигнал. Цей сигнал перериває виконання процесором послідовності команд, що задається виконуваним кодом, і спричиняє автоматичний перехід на заздалегідь визначену процедуру, названу процедурою оброблення переривань. У більшості моделей процесорів виконуваний апаратурою перехід на процедуру оброблення переривань супроводжується заміною «слова стану» машини (або навіть усього контексту процесу), що дозволяє одночасно з переходом за потрібною адресою виконати перехід у привілейований режим. Після завершення процесу оброблення переривань відбувається повернення до виконання перерваного коду.

Переривання є найважливішою функцією будь-якої ОС, будучи її рушійною силою. Дійсно, більша частина дій ОС ініціюється перериваннями різного типу. Навіть системні виклики від застосунків виконуються на багатьох апаратних платформах за допомогою спеціальної інструкції переривання, що викликає перехід до виконання відповідних процедур ядра (наприклад, інструкція int у процесорах Intel або SVC у мейнфреймах IBM).

Системний таймер, часто реалізований у вигляді швидкодіючого регістра-лічильника, необхідний ОС для витримування інтервалів часу. Для цього в регістр таймера програмно завантажується значення необхідного інтервалу в умовних одиницях, з якого потім автоматично з певною частотою починають відраховуватися одиниці. Частота «тіків» таймера, як правило, тісно пов'язана з частотою тактового генератора процесора. (Не слід плутати таймер ні з тактовим генератором, який виробляє сигнали, які синхронізують усі операції в комп'ютері, ні з системним годинником – працюючою на батареях електронною схемою, – які ведуть незалежний відлік часу й календарної дати.) Із досягненням нульового значення лічильника таймер ініціює переривання, яке обробляється процедурою ОС. Переривання від системного таймера використовуються ОС у першу чергу для спостереження за тим, як окремі процеси витрачають час процесора. Наприклад, у системі розділення часу під час оброблення чергового переривання від таймера планувальник процесів може примусово передати керування іншому процесу, якщо цей процес вичерпав виділений йому квант часу.

Засоби захисту ділянок пам'яті забезпечують на апаратному рівні перевірку можливості програмного коду здійснювати з даними певної ділянки пам'яті такі операції, як зчитування, запис або виконання (у разі передавання керування). Якщо апаратура комп'ютера підтримує механізм трансляції адрес, то засоби захисту ділянок пам'яті вмонтовуються в цей механізм. Функції апаратури із захисту пам'яті полягають у порівнянні рівнів привілеїв поточного коду процесора й сегмента пам'яті, до якого робиться звернення.

5.3 Машинозалежні компоненти ОС

Одна й та сама ОС не може без яких-небудь змін встановлюватися на комп'ютерах, що відрізняються типом процесора і/або способом організації всієї апаратури. У модулях ядра ОС не можуть не відобразитися такі особливості апаратної платформи, як кількість типів переривань і формат таблиці посилань на процедури оброблення переривань, склад регістрів загального призначення й системних регістрів, стан яких потрібно зберігати в контексті процесу, особливості підключення зовнішніх пристроїв і т. ін.

Однак досвід розроблення ОС показує: ядро можна спроектувати таким чином, що тільки деякі модулі будуть машинозалежними, а інші не будуть залежати від особливостей апаратної платформи. У добре структурованому ядрі машинозалежні модулі локалізовані й утворюють програмний шар, що природно примикає до шару апаратури, як це й показано на рис. 2.17. Така локалізація машинозалежних модулів суттєво спрощує перенесення ОС на іншу апаратну платформу.

Обсяг машинозалежних модулів ОС залежить від того, наскільки великі відмінності в апаратних платформах, для яких розробляться ОС. Наприклад, ОС, побудована на 64-бітових адресах, для перенесення на машину з 32-бітовими адресами має бути переписана наново. Одна з найбільш очевидних відмінностей – незбіжність системи команд процесорів – долається

достатньо просто. Операційна система програмується мовою високого рівня, а потім відповідним компілятором виробляється код для конкретного типу процесора. Однак у багатьох випадках відмінності в організації апаратури комп'ютера лежать набагато глибше й подолати їх у такий спосіб не вдається. Наприклад, однопроцесорний і двопроцесорний комп'ютери потребують застосування в ОС зовсім різних алгоритмів розподілу процесорного часу. Аналогічна відсутність апаратної підтримки віртуальної пам'яті приводить до принципової різниці в реалізації підсистеми керування пам'яттю. У таких випадках не обійтися без внесення в код ОС специфіки апаратної платформи, для якої ця ОС призначається.

Для зменшення кількості машинозалежних модулів виробники ОС звичайно обмежують універсальність машинонезалежних модулів. Це означає, що їх незалежність має умовний характер і поширюється тільки на кілька типів процесорів і створених на основі цих процесорів апаратних платформ. Цим шляхом пішли, наприклад, розробники ОС Windows NT, обмеживши кількість типів процесорів для своєї системи чотирма й поставляючи різні варіанти кодів ядра для однопроцесорних та багатопроцесорних комп'ютерів.

Особливе місце серед модулів ядра займають низькорівневі драйвери зовнішніх пристроїв. З одного боку, ці драйвери, як і високорівневі драйвери, входять до складу менеджера введення/виведення, тобто належать до шару ядра, що займає досить високе місце в ієрархії шарів. З другого боку, низькорівневі драйвери відображають усі особливості керованих зовнішніх пристроїв, тому їх можна віднести й до шару машинозалежних модулів. Така подвійність низькорівневих драйверів ще раз підтверджує схематичність моделі ядра із строгою ієрархією шарів.

Для комп'ютерів на основі процесорів Intel x86/Pentium розроблення екрануючого машинозалежного шару ОС дещо спрощується за рахунок вбудованої в постійну пам'ять комп'ютера базової системи введення-виведення – BIOS, яка містить драйвери для всіх пристроїв, що входять у базову конфігурацію комп'ютера: жорстких і гнучких дисків, клавіатури, дисплея і т. ін. Ці драйвери виконують досить примітивні операції з керованими пристроями, наприклад зчитування групи секторів даних з певної доріжки диска, але за рахунок цих операцій екрануються відмінності апаратних платформ ПК і серверів на процесорах Intel різних виробників. Розробники ОС можуть користуватися шаром драйверів BIOS як частиною машинозалежного шару ОС, або можуть замінити всі або частину драйверів BIOS компонентами ОС.

5.4 Переносимість ОС

Якщо код ОС може бути порівняно легко перенесений з процесора одного типу на процесор іншого типу й з апаратної платформи одного типу на апаратну платформу іншого типу, то таку ОС називають переносимою (portable), або мобільною.

Хоча ОС часто описуються або як переносимі, або як непереносимі, мобільність – це не бінарний стан, а поняття ступеня переносимості. Справа не в тому, чи може бути система перенесена, а в тому, наскільки легко можна це зробити. Для того щоб забезпечити властивість мобільності ОС, розробники повинні додержуватися таких правил.

1. Більша частина коду повинна бути написана мовою, транслятори якої є на всіх машинах, куди передбачається переносити систему. Такими мовами є стандартизовані мови високого рівня. Більшість переносимих ОС написано мовою С, яка має багато особливостей, корисних для розроблення кодів ОС, і компілятори якої широко доступні. Програма, написана мовою асемблеру, є переносимою тільки в тих випадках, коли перенесення ОС планується на комп'ютер, що має ту ж систему команд. В інших випадках асемблер використовується тільки для тих непереносимих частин системи, які повинні безпосередньо взаємодіяти з апаратурою (наприклад, обробник переривань), або для частин, які потребують максимальної швидкості (наприклад, цілочислова арифметика підвищеної точності).

2. Обсяг машинозалежних частин коду, які безпосередньо взаємодіють з апаратними засобами, має бути по можливості мінімізований. Так, наприклад, слід усіляко уникати прямого

маніпулювання регістрами й іншими апаратними засобами процесора. Для зменшення апаратної залежності розробники ОС повинні також унеможливити використання за замовчуванням стандартних конфігурацій апаратури або їх характеристик. Апаратно залежні параметри можна «сховати» у програмно-задавані дані абстрактного типу. Для виконання необхідних дій з керування апаратурою за цими параметрами повинен бути написаний набір апаратно залежних функцій. Щоразу, коли якому-небудь модулю ОС потрібно виконати якусь дію, пов'язану з апаратурою, він маніпулює абстрактними даними, використовуючи відповідну функцію з наявного набору. Коли ОС переноситься, то змінюються тільки ці дані й функції, які ними маніпулюють. Наприклад, в ОС Windows NT диспетчер переривань перетворює апаратні рівні переривань конкретного типу процесора в стандартний набір рівнів переривань IRQL, з якими працюють інші модулі ОС. Тому для перенесення Windows NT на нову платформу потрібно переписати, зокрема, ті коди диспетчера переривань, які займаються відображенням рівнів переривання на абстрактні рівні IRQL, а ті модулі ОС, які користуються цими абстрактними рівнями, не міняються.

3. Апаратно залежний код повинен бути надійно ізольований у декількох модулях, а не бути розподіленим по всій системі. Ізоляції підлягають усі частини ОС, які відображають специфіку як процесора, так і апаратну платформу в цілому. Низькорівневі компоненти ОС, що мають доступ до процесорно залежних структур даних і регістрів, мають бути оформлені у вигляді компактних модулів, які можна замінити аналогічними модулями для інших процесорів. Для усунення платформної залежності, що виникає через відмінності між комп'ютерами різних виробників, побудованими на тому самому процесорі (наприклад, MIPS R4000), повинен бути введений добре локалізований програмний шар машинозалежних функцій.

В ідеалі шар машинозалежних компонентів ядра повністю екранує іншу частину ОС від конкретних деталей апаратної платформи (кеш, контролери переривань введення-виведення і т. ін.), принаймні для того набору платформ, який підтримує ця ОС. У результаті відбувається підміна реальної апаратури якоюсь уніфікованою віртуальною машиною, однаковою для всіх варіантів апаратної платформи. Усі шари ОС, розміщені вище від шару машинозалежних компонентів, можуть бути написані для керування саме цією віртуальною апаратурою. Таким чином, у розробників з'являється можливість створювати один варіант машинонезалежної частини ОС (включаючи компоненти ядра, утиліти, системні обробні програми) для всього набору підтримуваних платформ.

6 Ресурси та їх класифікація

Ресурс, у загальному випадку – всякий споживаний продукт, який володіє деякою практичною цінністю для споживачів. Ресурси класифікують за такими ознаками.

За *реальністю існування*: фізичний і віртуальний. Під *фізичним* розуміють ресурс, який реально існує і при розподілі його між користувачами володіє всіма присутніми йому фізичними характеристиками. *Віртуальний ресурс* – це деяка модель фізичного ресурсу. Віртуальний ресурс не існує в тому вигляді, в якому він проявляє себе користувачу. Як модель віртуальний ресурс реалізується в деякій програмно-апаратній формі. У цьому розумінні віртуальний ресурс існує.

За *можливістю розширення властивостей*: еластичний і нееластичний (жорсткий). Характеризує ресурс з точки зору можливості побудови на його основі деякого віртуального ресурсу. Фізичний ресурс, який допускає "віртуалізацію", тобто відтворення і (або) розширення своїх властивостей, називають *еластичним*. *Нееластичним* називають фізичний ресурс, який за своїми внутрішніми властивостями не допускає віртуалізацію.

За *ступенем активності*: активний і пасивний. При використанні *активного ресурсу* він здатний виконувати дії по відношенню до інших ресурсів (або навіть по відношенню до самого себе) або процесів, які у загальному випадку приводять до зміни останніх. *Пасивний ресурс* не володіє такою властивістю. Над таким об'єктом проводять допустимі для нього дії, які можуть

привести до зміни його стану, тобто до зміни внутрішніх або зовнішніх характеристик. Центральний процесор (ЦП) – активний ресурс, а область пам'яті, яка виділяється на вимогу – пасивний ресурс.

За *часом існування*: постійний, тимчасовий. Якщо ресурс існує в системі до моменту породження процесу і доступний для використання за весь час інтервалу існування процесу, то такий ресурс є *постійним* для даного процесу. *Тимчасовий* ресурс може появлятися або знищуватися в системі динамічно за час існування розглядуваного процесу. Причому створення і зниження може здійснюватися як самим процесом, так і іншими процесами – системними або користувацькими. Ресурси можуть бути постійними для одних процесів та тимчасовими для інших.

За *ступенем важливості*: головний і другорядний. Ресурс є *головним* по відношенню до конкретному процесу, якщо без його виділення процес принципово не може розвиватися. До таких ресурсів відносяться перш всього ЦП і ОП. Ресурси, які допускають деяку альтернативу розвитку процесу, якщо вони не будуть виділені, називаються *другорядними*.

За *функціональною надлишковістю*: дорогі і дешеві. *Дорогий ресурс* надається швидко, але дорого. Дешевий ресурс надається з очікуванням.

За *структурою*: простий і складний. Структурний признак установлює наявність або відсутність у ресурсу деякої структури. Ресурс є *простим*, якщо не містить складових елементів і розглядується при розподілі як єдине ціле. *Складний ресурс* характеризується деякою структурою. Він містить у своєму складі ряд однотипних елементів, які володіють з точки зору користувачів, однаковими характеристиками. Простий і складний ресурси відрізняються числом станів. Простий ресурс може бути або "зайнятий", коли він виділений для використання якому-небудь процесу, або "вільний". Складний ресурс знаходиться в стані "вільний", якщо ні один з його складових елементів не розподілений для використання. Якщо всі елементи такого ресурсу виділені для використання, то він знаходиться в стані "зайнятий". Якщо частина елементів ресурсу розподілена, а решта ні, то ресурс "частково зайнятий".

За *відновлюваністю*: відновлювальний, спожитий. Вважається, що у відношенні кожного ресурсу процес користувача виконує три типи дій: ЗАПИТ, ВИКОРИСТАННЯ, ЗВІЛЬНЕННЯ. Якщо при розподілі системою ресурсу допускається багатократне виконання дій в послідовності запит-використання-звільнення, то такий ресурс називають *відновлювальним*. Після звільнення він стає доступним для іншого процесу. Якщо не враховувати зміни ресурсу при кожному разовому споживанні, то можна вважати час життя ресурсу достатньо великим. При зверненні до деякої категорії ресурсів використовується наступний порядок дій: звільнення-запит-використання, після чого ресурс вилучається із використання. Час життя такого *спожитого* ресурсу визначається періодом дій звільнення-споживання.

За *характером використання*: послідовний і паралельний. *Послідовний ресурс* допускає тільки послідовне в часі виконання ланцюжків дій "запит-виконання-звільнення" кожним процесом-споживачем цього ресурсу. *Паралельний ресурс* одночасно використовується більш ніж одним процесом. Паралельні процеси можуть звертатися до послідовно виконуваних ресурсів, які у такому випадку є критичними областями і повинні задовольняти правило взаємного виключення.

За *формою реалізації*: жорсткий і м'який ресурс. *Жосткі ресурси* не допускають копіювання і до них відносять апаратні компоненти машини, а також людські ресурси. Всі інші види ресурсів відносять до "*м'яких*". "Жорсткі" і "м'які" ресурси по різному втрачають і відновлюють роботоздатність. На відміну від "жорстких" "м'які" ресурси не втрачають роботоздатність з часом. Серед "м'яких" ресурсів виділяють два типи: програмні і інформаційні. Якщо "м'який" ресурс допускає копіювання і ефект від використання ресурсу-оригінала і ресурсу-копії ідентичний, то такий ресурс називають програмним м'яким ресурсом. В іншому випадку його відносять до інформаційного типу (це програми, файли, масиви і т.п.). "М'які" інформаційні ресурси або принципово не допускають копіювання, або допускають копіювання, яке є функцією часу. Це різного виду споживані ресурси: повідомлення, сигнали

переривань, запити до ОС, сигнали синхронізації. Такі повідомлення і сигнали інформаційно значимі на деякому скінченому інтервалі часу.

За *способом переміщення*: вивантажуваний і невивантажуваний. *Вивантажуваний ресурс* можна без наслідків забрати у володіючого ним процесу, наприклад, пам'ять. *Невивантажуваний ресурс* не можна забрати від власника, не знищивши результати обчислень. Наприклад, не можна перервати запис компакт-диску.

В термінах *операційної системи* (ОС) поняття ресурсу звичайно використовується по відношенню до повторно використовуваних, стабільних і часто невистачаючих об'єктів, які можуть запитуватися, використовуватися і звільнятися.

При розробленні перших ОС до ресурсів відносили процесорний час, пам'ять, канали введення-виведення і периферійні пристрої. З часом поняття ресурсу стало більш універсальним і загальним. До нього стали відносити і різного роду програмні і інформаційні ресурси, які з точки зору системи, також можуть бути об'єктами з можливостями розподілу і керування доступом. Поняття ресурсу перетворилося у абстрактну структуру з рядом атрибутів, які характеризують способи доступу до неї і її фізичне подання у системі. Крім системних ресурсів, в це поняття стали включати і такі об'єкти міжпроцесного обміну, як повідомлення і синхросигнали.

Одним із основних видів ресурсів є *процесор*. При цьому власне процесор як ресурс виступає тільки для багатопроцесорних систем, в однопроцесорних системах ресурсом є процесорний час.

Оперативна пам'ять є ресурсом, розподіл якого між процесами є актуальною задачею. У загальному випадку, власне пам'ять і доступ до неї є різними ресурсами.

Зовнішні пристрої є ресурсом, який при наявності механізмів прямого доступу можна використовувати одночасно. Пристрої, які мають тільки послідовний доступ, не є розподілюваними ресурсами.

Програмні модулі також є одним з ресурсів. Однократно виконувані модулі можуть бути правильно виконані один раз і тому вони є неділимим ресурсом. Повторно виконувані модулі можуть бути непривілейованими, привілейованими або з повторним входженням (реєнтерабельні).

Однократно виконуваними називають такі програмні модулі, які можна правильно виконати тільки один раз.

Повторно виконувані програмні модулі можуть бути непривілейованими, привілейованими і рентабельними.

Непривілейовані програмні модулі – це звичайні програмні модулі, що можуть бути перервані під час роботи. Отже, у загальному випадку їх не можна вважати розподілюваними, оскільки якщо після переривання виконання такого модуля в межах одного обчислювального процесу запустити його ще раз за вимогою іншого обчислювального процесу, тоді проміжні результати для перерваних обчислень можуть бути втрачені.

Привілейовані програмні модулі працюють у привілейованому режимі, тобто при відключеній системі переривань, тому ніякі зовнішні події не можуть порушити природний порядок обчислень. У результаті програмний модуль виконується до кінця, після чого він може бути знову викликаний для виконання іншого завдання.

Рентабельні програмні модулі допускають повторне багаторазове переривання виконання і повторний їх запуск за звертанням з інших завдань (обчислювальних процесів). Для цього рентабельні програмні модулі мають бути створені таким чином, щоб забезпечувалося збереження проміжних обчислень для обчислень, що перериваються, і повернення до них, коли обчислювальний процес відновлюється з перерваної раніше точки. Це може бути реалізовано двома способами: за допомогою статичних і динамічних методів виділення пам'яті для збережуваного значення. Основний, найбільш часто використовуваний динамічний спосіб, – виділення пам'яті для збереження всіх проміжних результатів обчислення, що належать до рентабельного програмного модуля.

Статичний спосіб виділення пам’яті полягає в такому: заздалегідь для фіксованої кількості обчислювальних процесів резервуються ділянки пам’яті, у яких розміщуватимуться змінні рентабельних програмних модулів: для кожного процесу є власна ділянка пам’яті. Найчастіше такі процеси є процесами введення/виведення, тобто йдеться про рентабельні драйвери.

Крім рентабельних програмних модулів є ще *повторно-вхідні*. Цим терміном називають програмні модулі, що теж допускають їх багаторазове виконання, але на відміну від рентабельних їх не можна переривати. Повторно-вхідні програмні модулі складаються з привілейованих секцій і повторне звернення до них можливе тільки після завершення якої-небудь з таких секцій. У повторно-вхідних програмних модулях чітко визначені всі можливі точки входів.

Дані виступають як інформаційний ресурс. Це змінні в ОП, файли. При використанні даних тільки для читання, вони легко розділяються. Якщо процеси змінюють інформаційний ресурс, то його розподілення значно ускладнюється.

Висновки.

• ОС виконує дві функції – розширювача машини і розподілювача апаратних ресурсів.

• ОС розрізняються особливостями реалізації внутрішніх алгоритмів керування основними ресурсами комп’ютера.

• ОС має багатошарову структуру: засоби апаратної підтримки, машинозалежні модулі, базові механізми ядра, менеджери ресурсів, інтерфейси системних викликів.

• Ядра ОС можуть працювати в захищеному або привілейованому режимі.

• До типових засобів апаратної підтримки ОС відносяться: підтримка привілейованого режиму, трансляція адрес, перемикання процесів, система переривань, системний таймер, захист ділянок пам’яті.

• Ресурс, у загальному випадку – всякий споживаний продукт, який володіє деякою практичною цінністю для споживачів. Ресурси класифікують за різними ознаками.

Література.

1. *Гордеев В.В., Молчанов А.Ю.* Системное программное обеспечение. – СПб.: Питер, 2001. – 736.

2. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.

Запитання

1. Наведіть визначення терміну ОС.
2. Які бувають ОС?
3. Яке призначення систем пакетної обробки?
4. Яке призначення систем розділення часу?
5. Яке призначення систем реального часу?
6. Наведіть визначення терміна однозадачна ОС.
7. Наведіть визначення терміна багатозадачна ОС.
8. Наведіть визначення терміна операційне середовище.
9. Наведіть визначення терміна ресурс.
10. Наведіть визначення терміна кластер.
11. Наведіть визначення терміна архітектура операційної системи.
12. Які бувають архітектури ОС?
13. Наведіть визначення терміна структура ОС.
14. Які бувають структури ОС?
15. Що таке системні оброблювальні програми?
16. Що таке програми надання користувачу додаткових послуг?
17. Що таке бібліотеки процедур різного призначення?
18. Що таке утиліти?
19. Назвіть типові засоби апаратної підтримки ОС та їх призначення.

3. СИСТЕМИ ДИНАМІЧНОГО ТА СТАТИЧНОГО ПЛАНУВАННЯ

Мета. Вивчення систем динамічного та статичного планування і диспетчеризації задач в комп'ютерних системах

Вступ. Можливість паралельного виконання процесів (потоків) залежить від кількості доступних процесорів. Якщо процесів (потоків) у системі більше, ніж доступних процесорів, ОС повинна розв'язувати задачу планування (sheduling). Головна мета планування для однопроцесорної системи полягає у такій організації виконання кількох процесів (потоків) на одному процесорі, за якої у користувача виникає враження, що вони виконуються одночасно.

При виконанні процесу він може знаходитися у різних станах. Керує станами і їх змінами спеціальна підсистема ОС. При плануванні процесів (потоків) розрізняють механізми і політики. Планування буває довготермінове, середньотермінове і короткотермінове (диспетчеризація). Дисципліни диспетчеризації поділяють на безпріоритетні і пріоритетні.

План.

1. Визначення етапів та структури процесу
2. Властивості та класифікація процесів
2.1 Контекст і дескриптор процесу
3. Загальні принципи планування
3.1. Механізми і політики планування
3.2. Види планування
4. Стратегії планування. Витісняльна і невитісняльна багатозадачність
5. Дисципліни короткотермінового планування (диспетчеризації)
6. Алгоритми планування
6.1. Планування за принципом FIFO
6.2. Кругове планування
6.3. Планування з пріоритетами
6.4. Планування на підставі характеристик подальшого виконання
6.5. Багаторівневі черги зі зворотним зв'язком
6.6. Лотерейне планування
6.7. Критерії порівняння алгоритмів диспетчеризації

1. Визначення етапів та структури процесу

Процес – це система дій, яка реалізує певну функцію в обчислювальній системі і оформлена так, що керуюча програма обчислювальної системи може перерозподіляти ресурси цієї системи з метою забезпечення багатопрограмного режиму роботи.

Ця система дій може відбуватися по різному. Програмі в деякий момент часу може бути наданий процесор для виконання або можуть бути надані результати роботи інших процесів або інших ресурсів. Одним словом, ходом виконання програми (процесу) необхідно керувати. Функція системних програм в загальному випадку, а операційної системи зокрема – забезпечення ефективного керування процесами, як по відношенню до кожного процесу, так і до їх сукупності.

При виконанні програм на процесорі обчислювальної системи визначають наступні стани процесу:

- створення (new)– процес завантажується в пам'ять, але його виконання ще неініційоване (пасивний стан).

- готовність (ready) – процес готовий до виконання, має всі ресурси і очікує ресурс центрального процесора (пасивний стан).

- виконання (running) – інструкції процесу виконує центральний процесор (активний стан).

- очікування (waiting) – процес очікує деяку подію (наприклад, завершення операції введення-виведення) (пасивний стан). Такий стан називається заблокований, а процес – призупинений.

- завершення (terminated) – потік завершив виконання (якщо при цьому його ресурси не були звільнені, то він переходить у додатковий стан – зомбі).

В залежності від типу очікуваного ресурсу розрізняють наступні системи:

- з обмеженням на введення/виведення (в таких системах найбільший час очікування витрачається процесом при зверненні до підсистем введення/виведення);

- з обмеженням по швидкодії (найбільший час витрачається на доступ до процесора).

Стани процесів і умови їх зміни показані на рис. 1.

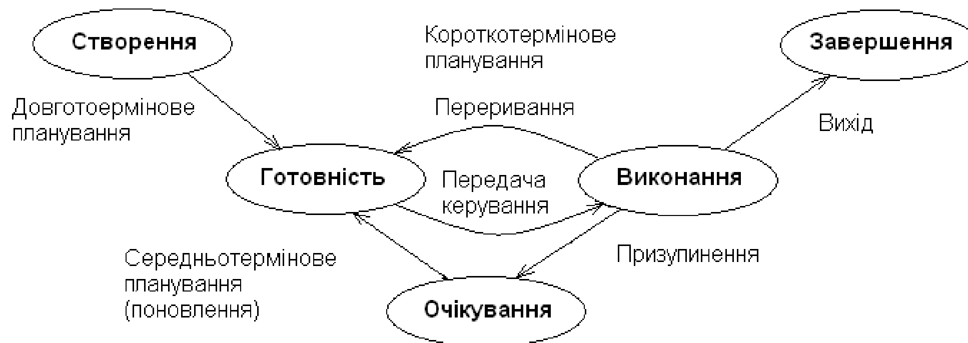

Рисунок 1 – Стани процесів і умови їх зміни

У загальному випадку процес складається з:

- коду процесу;
- підсистеми керуванням процесу;
- підсистеми даних.

Код процесу – це його ідентифікаційний номер для його ідентифікації в ОС.

Підсистема керування процесом – містить наступну інформацію:

- символьний ідентифікатор процесу – це символьне ім’я процесу, яке присвоюється йому системою – планувальником і виконується для визначення стану процесу;
- пріоритет процесу – встановлюється як системним планувальником так і користувачем, може бути змінений диспетчером процесів функціонування;
- посилання на попередній процес в черзі процесу і на наступний;
- ідентифікатор поточного стану процесу;
- квант процесорного часу наданий даному процесу.

Стек процесу може бути розміщений як в підсистемі керування процесу, так і підсистемі даних. В будь-якому випадку адреса верхівки стеку та його місткість відомі з метою збереження поточного стану процесу при перериванні його виконання.

Підсистема даних – містить вказівники на зовнішні змінні (семафори, прапорці, зовнішні буфери введення/виведення), внутрішні змінні, опис спеціалізованих пристроїв, які можуть бути використані даним процесом.

2. Властивості та класифікація процесів

Процеси класифікують за наступними ознаками:

Таблиця 1 – Класифікація процесів

За належністю до ЦП	внутрішні зовнішні

За належністю до ОС	системні користувацькі
За генеалогічним типом	породжуючі породжені
За результативністю	рідні еквівалентні поточні рівні
За динамічним типом	послідовні паралельні комбіновані
За зв'язністю	ізольовані взаємодіючі конкуруючі інформаційно-незалежні

Процеси реального часу - це процеси, які потребують такого планування щоб гарантувати закінчення процесу до певного моменту часу.

Інтерактивні – це процеси, час існування яких повинен бути не більший ніж інтервал часу допустимої реакції ЕОМ на запити користувачів.

Породжуючі – це процеси, які можуть породжувати існування інших процесів.

Процеси, які починають існувати в результаті існування іншого процесу називаються породженими. Коли процеси породжуючі і породжені, то вони називаються комбінованими.

Траса – це тривалість і порядок перебування процесу в допустимих станах на інтервалі існування.

Два процеси, які мають один і той же результат обробки даних в одній і тій же програмі на одному і тому ж, або на різних процесорах називаються *еквівалентними*. Траси еквівалентних процесів не співпадають. Якщо в кожному з еквівалентних процесів обробка даних проходить в одній і тій же програмі, але траси не співпадають, то такі процеси називаються *тотожними*. При співпадінні трас тотожних процесів їх називають *рівними*. Якщо інтервали двох процесів не пересікаються в часі, то такі процеси називаються *послідовними*. Якщо на певному інтервалі часу існують одночасно два процеси, то вони називаються *паралельними*. Якщо на певному інтервалі знайдеться хоча б одна точка в якій існує процес, але не існує інший і хоча б одна точка в якій ці два процеси існують одночасно, то ці процеси називаються *комбінованими*.

В операційних системах прийнято розрізняти не тільки час існування процесу, але й час його народження. Такою точкою відліку прийнято вважати ЦП. Процеси виконані на ньому називаються програмними або внутрішніми.

Зовнішні процеси – це процеси, розвиток яких проходить не під контролем ЦП.

Програмні процеси – поділяються на *системні* і *користувацькі*. При розвитку системного процесу виконується програма із складу операційної системи. При розвитку користувацького – програма користувача. Два процеси називаються *взаємозв'язаними* якщо між ними створюються зв'язки за допомогою системи управління процесів. В іншому випадку вони називаються *ізольованими.* Якщо два взаємозв'язані процеси використовують одні і ті ж ресурси, але не обмінюються між собою інформацією, то вони називаються *інформаційно-незалежними*. Якщо між двома процесами є інформаційні зв'язки, то вони називаються *взаємодіючими*.

2.1 Контекст і дескриптор процесу

За час існування процесу його виконання може бути багаторазово перерване і продовжене. Для того, щоб відновити виконання процесу, необхідно відновити стан його операційного середовища. Стан операційного середовища відображається станом регістрів і програмного

лічильника, режимом роботи процесора, вказівниками на відкриті файли, інформацією про незавершені операції введення/виведення, кодами помилок виконуваних процесом системних викликів і т. ін. Цю інформацію називають *контекстом процесу*.

Крім цього, ОС для реалізації планування процесів потрібна додаткова інформація про ідентифікатор процесу, стан процесу, дані про ступінь привілейованості процесу, місце перебування кодового сегмента й інша інформація. У деяких ОС (наприклад, в ОС UNIX) інформацію такого роду, використовувану ОС для планування процесів, називають *дескриптором процесу*. Дескриптор процесу порівняно з контекстом містить більш оперативну інформацію, яка має бути легко доступна підсистемі планування процесів. Контекст процесу містить менш актуальну інформацію і використовується ОС тільки після прийняття рішення про поновлення перерваного процесу.

Черги процесів є дескрипторами окремих процесів, об'єднаних в списки. Таким чином, кожен дескриптор, крім всього іншого, містить принаймні один вказівник на інший дескриптор, що перебуває з ним у черзі. Така організація черг дозволяє легко їх перевпорядковувати, включати та виключати процеси, переводити процеси з одного стану в інший.

Програмний код тільки тоді почне виконуватися, коли для нього ОС буде створений процес. Створити процес – це означає:

1) створити інформаційні структури, що описують цей процес, тобто його дескриптор і контекст;

2) включити дескриптор нового процесу в чергу готових процесів;

3) завантажити кодовий сегмент процесу в оперативну пам'ять чи в область гортання сторінок.

3. Загальні принципи планування

З погляду планування виконання потоку можна зобразити як цикл чергування періодів обчислень (використання процесора) і періодів очікування введення-виведення. Інтервал часу, упродовж якого потік виконує тільки інструкції процесора називають інтервалом використання процесора (CPU burst), інтервал часу, коли потік очікує введення-виведення, – інтервалом введення-виведення (I/O burst). Найчастіше ці інтервали мають тривалість від 2 до 8 мс.

Потоки, які більше часу витрачають на обчислення, називаються обмеженими можливостями процесора (CPU bound). Потоки, які більшу частину часу перебувають в очікуванні введення-виведення, називаються обмеженими можливостями введення-виведення (I/O bound).

3.1. Механізми і політики планування

Слід розрізняти *механізми* і *політики* планування. До механізмів планування належать засоби перемикання контексту, засоби синхронізації потоків тощо, до політики планування – засоби визначення моменту часу, коли необхідно перемкнути контекст. Та частина системи, яка відповідає за політику планування, називають *планувальником* (sheduler), а алгоритм, що виконується при цьому, – *алгоритмом планування* (scheduling algorithm).

Для оцінки політики планування використовуються наступні критерії:

• Мінімальний час відгуку між запуском потоку і отриманням першої відповіді. Для сучасних систем прийнятним часом відгуку є 50-150 мс.

• Максимальна пропускну здатність – кількість задач, які система може виконувати за одиницю часу.

• Справедливість – виділення процесорного часу потокам відповідно до їхньої важливості.

3.2. Види планування

Планування – це системний процес, який ставить процеси користувачів в чергу та визначає атрибути їх виконання в рамках використовуваної обчислювальної системи.

Розрізняють планування довготермінове (long-term scheduling), середньотермінове (medium-term scheduling) і короткотермінове (short-term scheduling).

Засоби *довготермінового планування* визначають, яку з програм треба завантажити у пам’ять. Таке планування називають також *статичним*, оскільки воно не залежить від поточного стану системи.

Засоби *середньотермінового планування* керують переходами потоків із стану очікування в стан готовності й назад.

Для коректної організації очікування, крім черги готових потоків, реалізують додатковий набір черг очікування. Середньотерміновий планувальник керує всіма цими чергами.

Короткотермінове планування (*диспетчеризація*), або планування процесора (CPU scheduling) дає змогу відповісти на два базових запитання:

- коли перервати виконання потоку;
- якому потокові з числа готових до виконання потрібно передати процесор у поточний момент.

Середньотермінове і короткотермінове планування називають також *динамічним*, оскільки рішення приймаються під час функціонування ОС на основі аналізу поточної ситуації.

Короткотерміновий планувальник – це підсистема ОС, яка в разі необхідності перериває активний потік і вибирає з черги готових потоків той, що має виконуватися.

Усі стратегії й алгоритми планування, які будуть далі розглядуватися, належать до короткотермінового планування.

4. Стратегії планування. Витісняльна і невитісняльна багатозадачність

В загальному випадку стратегія планування визначає, які процеси повинні бути заплановані на виконання для отримання результатів вирішуваної задачі. Стратегії залежать від варіанту передачі керування від одного потоку до іншого:

- після того, як потік перейшов у стан очікування (наприклад, під час введення-виведення або приєднання);
- після закінчення виконання потоку;
- явно (коли потік сам віддає процесор іншим потокам);
- за перериванням (наприклад, вид таймера).

Системний процес який здійснює планування може бути реалізований двома способами:

- *цілісний планувальник* – це програмний модуль, який є частиною операційної системи, його робота ініціюється перериваннями і він виконується як окремий системний процес;
- *розподілений планувальник* – програмний модуль планувальника або його частина записується в кожний процес при його завантаженні в оперативну пам’ять.

У загальному випадку планувальник здійснює наступні функції:

- запис копії процесу в пам’ять обчислювальної системи;
- аналіз атрибутів при виконанні обчислювального процесу та формування набору робочих атрибутів в залежності від характеристик обчислювальної системи;
- запис процесу в чергу процесів у випадку, якщо є вільне місце в черзі, або переведення процесу в пасивний стан у випадку, якщо всі елементи черги зайняті;
- періодичний перегляд черги процесів, запис або знищення процесів в черзі у відповідності з часом функціонування процесів.

В залежності від використання планувальника розрізняють наступні стратегії планування потоків: витісняльна і невитісняльна багатозадачність.

При *витісняльній багатозадачності* (preemptive multitasking) потоки, що логічно мають виконуватися, можуть бути тимчасово перервані планувальником ОС без їхньої участі для передачі керування іншим потокам. Переривання виконання потоку й передача керування

іншому потокові найчастіше здійснюють в обробнику переривання від системного таймера. Така стратегія реалізована в усіх сучасних ОС.

При *невитіснянльній багатозадачності* (non-preemptive multitasking) потоки можуть виконуватися упродовж необмеженого часу й не можуть бути перервані ОС, а також повинні самі віддавати керування ОС для передачі іншим потокам або, принаймні, переходити у стан очікування.

5. Дисципліни короткотермінового планування (диспетчеризації)

Розрізняють два великих класи дисциплін короткотермінового планування (диспетчеризації): безпріоритетні і пріоритетні.

При *безпріоритетній* організації задачі або процеси вибираються згідно з певним, раніше установленим, порядком без врахування важливості цих процесів та необхідного часу обслуговування.

При реалізації *пріоритетних* дисциплін певним окремим задачам чи процесам надаються виключне право або набір прав для поміщення такого процесу в чергу.

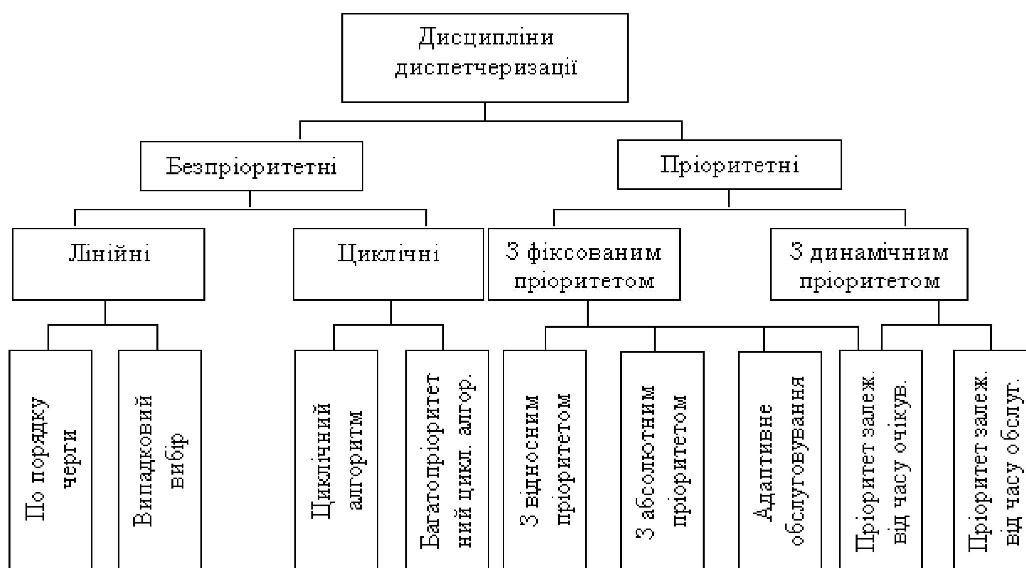

Рисунок 2 – Види короткотермінового планування (диспетчеризації)

Розглянемо деякі дисципліни короткотермінового планування:

1) *Дисципліна FIFO* (first input first output) – є реалізацією безпріоритетної черги. Якщо в черзі звільняється елемент і він не останній, то всі після нього посуваються вперед і в останній вільний елемент черги записується ім'я нового процесу. Першим з черги вибирається елемент, який стоїть на початку черги.

2) *Дисципліна LIFO* (last input first output) – першим з черги вибирається елемент, який прийшов останнім.

3) Дисципліна FCFS (first come first serve) – є подібна до FIFO. Ця дисципліна враховує перебування процесів в станах блокування, наприклад при операціях вводу/виводу. Ті, які були заблоковані в процесі роботи після переходу в стан готовності поступають в чергу готовності перед новими задачами, які ще не обслуговувались. При такій дисципліні організовуються дві черги: черга готових до обслуговування процесів і черга нових процесів.

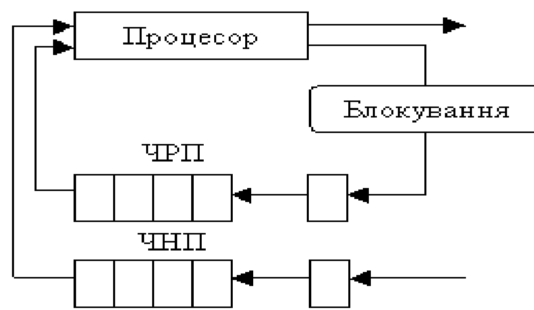

Рисунок 3 – Дисципліна FCFS: ЧРП – черга робочих процесів, ЧНП – черга нових процесів

Ця дисципліна не потребує втручання ззовні в хід обчислювального процесу. Не відбувається розподіл процесорного часу. Воно не відноситься до дисциплін без виключення. Переваги – простота реалізації та невеликий розхід системних ресурсів для організації черги задачі. Однак при збільшенні завантаженості ОС, збільшується середній час обслуговування задачі, коли "короткі" завдання (які потребують невеликих затрат машинного часу) повинні очікувати такий самий час, що й довгі завдання.

4) *Дисципліна SJN* (short jump next) – згідно з цією дисципліною ОС вимагає, щоб для кожного процесу була відома оцінка за необхідними обчислювальними ресурсами. Для постановки задачі в чергу диспетчер оцінює необхідний час виконання задачі і ставить задачу перед довшою задачею. При цій дисципліні існує одна черга процесів і завдання, що були заблоковані знову поступають в кінець черги як і нові завдання. Це приводить до того, що завдання які потребують мало часу очікують в черзі процесів як і довгі процеси.

5) *Дисципліна SRT* (short remain time) – розроблена з метою забезпечення більш якісного обслуговування коротких завдань. Виконує спочатку ті процеси, час завершення виконання яких найменший.

6) *Дисципліна RR* (round robin) – кожна задача отримує порцію часу (квант часу). Після закінчення цього кванту часу задача знімається з виконання на процесорі і він передається наступній задачі. Задача, яка була знята з черги записується в кінець черги задач, які готові до виконання. Величина кванту часу, як правило, вибирається, як середнє значення між достатнім часом реакції системи на запити користувачів та процесорним часом, який необхідний для перемикання між задачами.

6. Алгоритми планування

Алгоритм планування дає змогу короткотерміновому планувальнику вибирати з готових до виконання потоків той, котрий потрібно виконувати наступним. Можна сказати, що алгоритми планування реалізовують політику планування.

6.1. Планування за принципом FIFO

Найпростіший невитісняльний алгоритм, у якому потоки ставлять на виконання в порядку їхньої появи у системі до переходу в стан очікування й виконують до переходу в стан очікування, явної передачі керування або завершення. Чергу готових потоків при цьому організовують за принципом FIFO.

Як тільки в системі створюється новий потік, його керуючий блок додається у хвіст черги. Коли процесор звільняється, його надають потоку з голови черги.

6.2. Кругове планування

Найпростішим витісняльним алгоритмом є алгоритм *кругового планування* (round-robin sheduling). Кожному потоку виділяють інтервал часу, який називається квантом часу (time

quantum, time slice), і упродовж якого цьому потокові дозволено виконуватися. Коли потік усе ще виконується після вичерпання кванту часу, його переривають і перемикають процесор на виконання інструкцій іншого потоку. Коли він блокується або закінчує своє виконання до вичерпання кванту часу, процесор теж передають іншому потокові. Довжина кванту часу для всієї системи однакова.

Алгоритм реалізується як черга готових потоків на основі циклічного списку. Коли потік вичерпав свій квант часу, його переміщають у кінець списку, туди ж додають і нові потоки. Перевірку вичерпання кванту часу виконують в обробнику переривання від системного таймера.

При круговому плануванні припускається, що всі потоки однаково важливі. Єдиною характеристикою, що впливає на роботу алгоритму, є довжина кванту часу.

6.3. Планування з пріоритетами

Якщо потоки мають різну важливість то застосовують **планування з пріоритетами**. При цьому кожному потокові надають пріоритет і на виконання ставиться потік із найвищим пріоритетом із черги готових потоків. Пріоритети потокам можуть надаватися статично або динамічно.

Одним із підходів до реалізації планування із пріоритетами є алгоритм багаторівневих черг (multilevel queues). У цьому випадку організують кілька черг для груп потоків із різними пріоритетами.

Рішення про вибір потоку для виконання приймають наступним чином:

- якщо в черзі потоків із найвищим пріоритетом є потоки, для них слід використати якийсь простіший алгоритм планування (наприклад, кругового планування);
- якщо в черзі немає жодного потоку, переходять до черги потоків з нижчими пріоритетами.

Для різних черг можна використати різні алгоритми планування, крім того, кожній черзі може бути виділена певна частка процесорного часу.

6.4. Планування на підставі характеристик подальшого виконання

Важливим класом алгоритмів планування з пріоритетами є алгоритми, в яких рішення про вибір потоку для виконання приймають на підставі знання або оцінки характеристик його подальшого виконання.

В алгоритмі "*перший – із найкоротшим часом виконання*" (Shortest Time to Completion First, STCF), з кожним потоком пов'язують тривалість наступного інтервалу використання ним процесора і для виконання щоразу вибирають потік з найкоротшим таким інтервалом.

Для короткотермінового планування може бути реалізоване наближення до цього алгоритму, засноване на оцінці довжини чергового інтервалу використання процесора з урахуванням попередніх інтервалів того самого потоку. Для обчислення цієї оцінки можна використати рекурсивну формулу:

$$t_{n+1} = \alpha T_n + (1-\alpha)t_n,\ 0 \le \alpha \le 1,\ t_0 = T_0,$$

де t_{n+1} – оцінка довжини інтервалу; t_n – оцінка довжини попереднього інтервалу; T_n – справжня довжина попереднього інтервалу. Найчастіше використовують $\alpha = 0{,}5$, у цьому разі для перерахунку оцінки достатньо обчислити середнє між попередньою оцінкою і реальним значенням інтервалу.

Витісняльним аналогом STCF є алгоритм "*перший – із найкоротшим часом виконання, що залишився*" (Shortest Remaining Time to Completion First, SRTCF). Його відмінність від STCF полягає в тому, що, коли в чергу готових до виконання потоків додають новий, у якого наступний інтервал використання процесора коротший, ніж час, що залишився до завершення виконання, поточний потік переривається, і на його місце стає новий потік.

6.5. Багаторівневі черги зі зворотним зв’язком

Алгоритм *багаторівневих черг зі зворотним зв’язком* (multilevel feedback qurues) є найуніверсальнішим алгоритмом планування, але при цьому одним із найскладніших у реалізації.

З погляду організації структур даних цей алгоритм схожий на звичайний алгоритм багаторівневих черг: є кілька черг готових потоків із різним пріоритетом, при цьому потоки черги із нижчим пріоритетом виконуються, тільки коли всі черги верхнього рівня порожні.

Відмінності між двома алгоритмами полягають у тому, що:

- потокам дозволено переходити з рівня на рівень (із черги в чергу);

- потоки в одній черзі об’єднуються не за пріоритетами, а за довжиною інтервалу використання процесора, потоки з коротшим інтервалом перебувають у черзі з більшим пріоритетом.

Усередині всіх черг, крім найнижчої, використовують кругове планування (у найнижчій працює FIFO алгоритм). Різні черги відповідають різній довжині кванта часу – що вищий пріоритет, то коротший квант (звичайно довжина кванту для сусідніх черг зменшується удвічі). Якщо потік вичерпав свій квант часу, він переміщається у хвіст черги із нижчим пріоритетом. У результаті потоки з коротшими інтервалами залишаються з високим пріоритетом, а потоки з довшими інтервалами подовжують свій квант часу (рис. 1). Можна також автоматично переміщати потоки, які давно не отримували керування, із черги нижнього рівня на рівень вище.

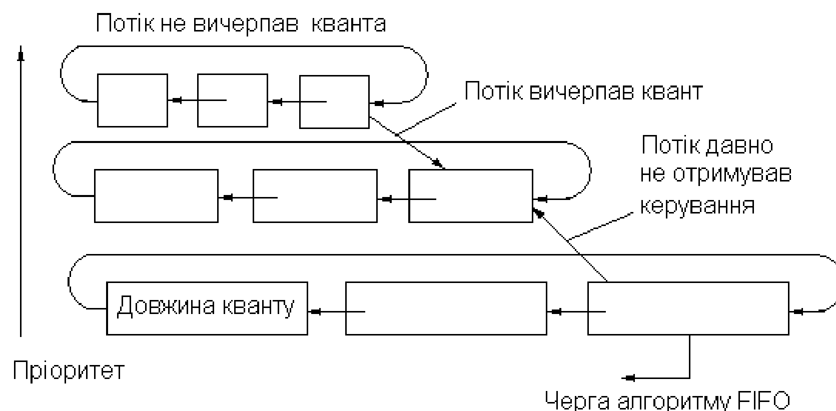

Рисунок 4 – Багаторівневі черги зі зворотним зв’язком

6.6. Лотерейне планування

Алгоритм *лотерейного планування* (lottery scheduling) є простим для розуміння і легким для реалізації, але разом з тим має великі можливості.

Ідея лотерейного планування полягає в наступному:

- потік отримує деяку кількість лотерейних квитків, кожен з яких дає право користуватися процесором упродовж часу Т;

- планувальник через проміжок часу Т вибирає випадково один лотерейний квиток;

- потік, “що виграв”, дістає керування до наступного розіграшу;

Лотерейне планування дає змогу:

- емулювати кругове планування, видавши кожному потокові однакову кількість квитків;

- емалювати планування з пріоритетами, розподіляючи квитки відповідно до пріоритетів потоків;

- емалювати SRTCF – давати коротким потокам більше квитків, ніж довгим (якщо потік отримав хоча б один квиток, він не простоюватиме);

- забезпечити розподіл процесорного часу між потоками – дати кожному потокові кількість квитків, пропорційну до частки процесорного часу, який потрібно йому виділити;
- динамічно міняти пріоритети, відбираючи і додаючи квитки в процесі роботи.

6.7. Критерії порівняння алгоритмів диспетчеризації

Існують такі критерії порівняння алгоритмів диспетчеризації**:**

- використання (завантаження) ЦП – для більшості ПК середня завантаженість процесора не перевищує 3%, однак в моменти складних обчислень завантаженість процесора може досягати 100%. В системах, де ПК виконує багато роботи (сервер), завантаженість – 15-40%, може доходити до 100% при повному завантаженні;
- пропускна здатність процесора – це кількість процесів, яка виконується процесором за одиницю часу;
- час обороту – інтервал часу від моменту появи процесу у вхідній черзі до моменту завершення процесу. Цей час обороту включає в себе час очікування у вхідній черзі, час очікування у черзі готовності, час готовності у чергах до периферійних пристроїв, час виконання на процесорі та час введення/виведення;
- час очікування – це сумарний час знаходження процесу в черзі очікування готових процесів;
- час відповіді – це час від моменту поступлення процесу на вхідну чергу до моменту першого звернення процесу на ввід/вивід даних.

Висновки.

• Процес – це система дій, яка реалізує певну функцію в обчислювальній системі.

• Процес може знаходитися в одному з наступних станів: завантаження, готовності, виконання, очікування.

• У загальному випадку процес складається з коду процесу, підсистеми керуванням процесу, підсистеми даних.

• Для організації виконання процесів у обчислювальній системі використовуються різні стратегії планування – обслуговування процесів згідно черги, надання переваги менш тривалим процесам, рівномірне обслуговування всіх процесів.

• Системний процес, який здійснює планування називається планувальником.

• Для планування процесів використовують алгоритми побудовані за різними принципами – принципом FIFO, круговим, з пріоритетами, з використанням характеристик подальшого виконання, лотерейний.

Література.

1. *Гордеев В.В., Молчанов А.Ю.* Системное программное обеспечение. – СПб.: Питер, 2001. – 736.
2. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.

Запитання.

1. Дати визначення процесу.
2. Стани і умови зміни станів.
3. Ознаки класифікації процесів.
4. Контекст і дескриптор процесу.
5. Механізми і політики планування.
6. Види і стратегії планування.
7. Дисципліни короткотермінового планування (диспетчеризації).
8. Алгоритми кругового планування.
9. Алгоритм планування з пріоритетами.
10. Алгоритм планування на підставі характеристик подальшого виконання.

11. Алгоритм планування - багаторівневі черги зі зворотним зв’язком.
12. Алгоритм лотерейного планування.
13. Критерії порівняння алгоритмів диспетчеризації.

4. ОРГАНІЗАЦІЯ ОПЕРАТИВНОЇ ПАМ’ЯТІ

Мета. Вивчення організації і керування оперативною пам’яттю в ОС.

Вступ. Під пам’яттю розуміють ресурс комп’ютера, призначений для зберігання програмного коду і даних. Пам’ять зображають як масив машинних слів або байтів з їхніми адресами. У фон-нейманівській архітектурі комп’ютерних систем процесор вибирає інструкції і дані з пам’яті та може зберігати в ній результати виконання операцій.

Різні види пам’яті організовані в ієрархію. На нижніх рівнях такої ієрархії перебуває дешевша і повільніша пам’ять більшого обсягу, а в міру просування ієрархією вверх пам’ять стає дорожчою і швидшою (а її обсяг стає меншим). Найдешевшим і найповільнішим запам’ятовувальним пристроєм є *жорсткий диск* комп’ютера. Його називають також допоміжним запам’ятовувальним пристроєм (secondary storage). Швидшою й дорожчою є *оперативна пам’ять*, що зберігається в мікросхемах пам’яті, встановлених на комп’ютері, – таку пам’ять називають основною пам’яттю (main memory). Ще швидшими засобами зберігання даних є різні *кеші* процесора, а обсяг цих кешів ще обмеженіший.

План.

1 Поняття віртуальної пам’яті
2 Фрагментація пам’яті
3 Логічна і фізична адресація пам’яті
4 Базовий і межовий регістр
5 Сегментація пам’яті
6. Сторінкова організація пам’яті
6.1 Багаторівневі таблиці сторінок
6.2 Реалізація таблиць сторінок в архітектурі IA_32
7 Асоціативна пам’ять
8 Сторінково-сегмента організація пам’яті

1 Поняття віртуальної пам’яті

Віртуальна пам’ять — це технологія, в якій вводиться рівень додаткових перетворень між адресами пам’яті, використовуваних процесом, і адресами фізичної пам’яті комп’ютера. Такі перетворення мають забезпечувати захист пам’яті та відсутність прив’язання процесу до адрес фізичної пам’яті.

Завдяки віртуальній пам’яті фізична пам’ять адресного простору процесу може бути фрагментованою, оскільки основний обсяг пам’яті, яку займає процес, більшу частину часу залишається вільним. Є так зване правило «дев’яносто до десяти», або правило локалізації, яке стверджує, що 90 % звертань до пам’яті у процесі припадає на 10% його адресного простору. Адреси можна переміщати так, щоб основній пам’яті відповідали тільки ті розділи адресного простору процесу, які справді використовуються у конкретний момент.

При цьому невикористовувані розділи адресного простору можна ставити у відповідність повільнішій пам’яті, наприклад простору на жорсткому диску, а в цей час інші процеси можуть використовувати основну пам’ять, у яку раніше відображалися адреси цих розділів. Коли ж розділ знадобиться, його дані завантажують з диска в основну пам’ять, можливо, замість розділів, які стали непотрібними в конкретний момент (і які, своєю чергою, тепер збережуться на диску). Дані можуть зчитуватися з диска в основну пам’ять під час звертання до них.

У такий спосіб можна значно збільшити розмір адресного простору процесу і забезпечити виконання процесів, що за розміром перевищують основну пам’ять.

Одним із варіантів відображення простору імен на фізичну пам’ять є відповідність віртуального адресного простору фізичній пам’яті. При цьому немає необхідності здійснювати

друге повторне відображення. В даному випадку система програмування генерує абсолютну двійкову програму. Така програма може виконатись тільки в тому випадку, якщо її віртуальні адреси будуть точно відповідати фізичним.

Частина програмних модулів будь-якої ОС повинна бути абсолютно двійковими програмами. Ці програми розміщуються за фіксованими адресами і за їх допомогою розміщують інші програм за різними фізичними адресами.

Інший варіант відображення здійснюється завантажувачем (loader). Після завантаження програми віртуальні адреси втрачаються і доступ здійснюється безпосередньо до фізичних комірок. Схеми варіантів відображення показані на рис. 1.

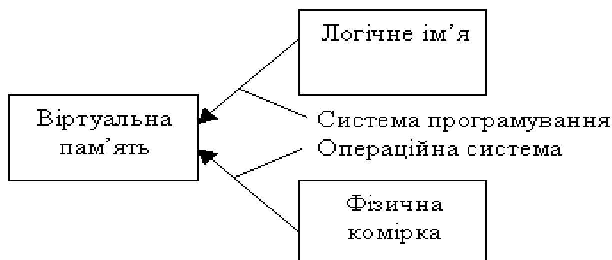

Рисунок 1 – Схема відображень пам'яті

2 Фрагментація пам'яті

Фрагментація пам'яті – це ситуація, коли неможливо використати вільну пам'ять. Розрізняють зовнішню і внутрішню фрагментацію пам'яті (рис. 2).

Зовнішня фрагментація зводиться до того, що внаслідок виділення і наступного звільнення пам'яті в ній утворюються вільні блоки малого розміру – дірки (holes). Через це може виникнути ситуація, за якої неможливо виділити неперервний блок пам'яті розміру N, оскільки немає жодного неперервного вільного блоку, розмір якого S>N, хоча загалом обсяг вільного простору пам'яті перевищує N. Так, на рис. 2 для виконання процесу P5 місця через зовнішню фрагментацію не вистачає.

Внутрішня фрагментація зводиться до того, що за запитом виділяють блоки пам'яті більшого розміру, ніж насправді будуть використовуватися, у результаті всередині виділених блоків залишаються невикористані ділянки, які вже не можуть бути призначені для чогось іншого.

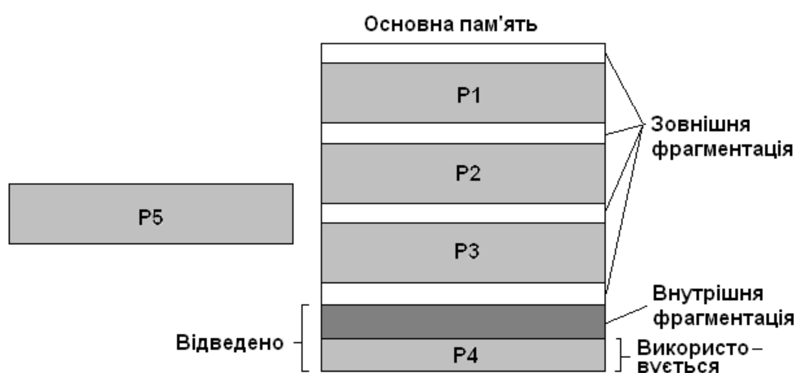

Рисунок 2 – Зовнішня і внутрішня фрагментація

3 Логічна і фізична адресація пам'яті

Найважливішими поняттями концепції віртуальної пам'яті є логічна і фізична адресація пам'яті.

Логічна або віртуальна адреса – адреса, яку генерує програма, запущена на деякому процесорі. Адреси, що використовують інструкції конкретного процесора, є логічними адресами. Сукупність логічних адрес становить логічний адресний простір.

Фізична адреса — адреса, якою оперує мікросхема пам'яті. Прикладна програма в сучасних комп'ютерах ніколи не має справи з фізичними адресами. Спеціальний апаратний пристрій MMU (memory management unit – пристрій керування пам'яттю) відповідає за перетворення логічних адрес у фізичні. Сукупність усіх доступних фізичних адрес становить фізичний адресний простір. Отже, якщо в комп'ютері є мікросхеми на 1024 МБайт пам'яті, то саме такий обсяг пам'яті адресують фізично. Логічно зазвичай адресують значно більше пам'яті.

Найпростіша схема перетворення адрес зображена на рис. 3.

Рисунок 3 – Зовнішня і внутрішня фрагментація

4 Базовий і межовий регістр

Під час реалізації віртуальної пам'яті необхідно забезпечити захист пам'яті, переміщення процесів у пам'яті та спільне використання пам'яті кількома процесами.

Одним із найпростіших способів задовольнити ці вимоги є використання *базового* і *межового регістрів*. Для кожного процесу в двох регістрах процесора зберігають два значення – базової адреси (base) і межі (bounds). Кожний доступ до логічної адреси апаратно перетворюється у фізичну адресу шляхом додавання логічної адреси до базової. Якщо отримувана фізична адреса не потрапляє в діапазон (base, base+bounds), вважають, що адреса невірна, і генерують помилку (рис. 4).

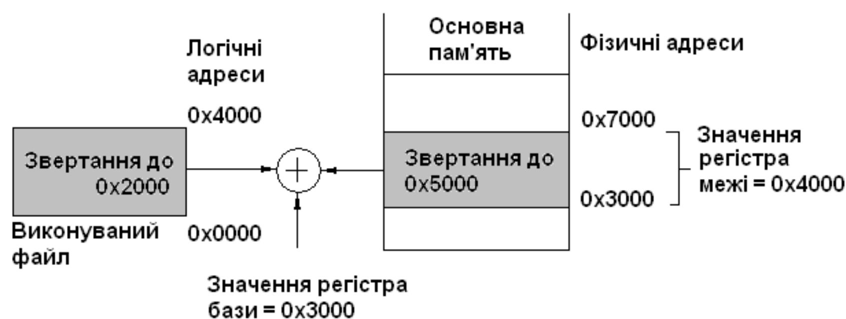

Рисунок 4 – Використання базового і межового регістрів

Такий підхід є найпростішим прикладом реалізації динамічного переміщення процесів у пам'яті. Усі інші підходи є різними варіантами розвитку цієї базової схеми. Наприклад, те, що кожний процес у разі використання цього підходу має свої власні значення базового і межового регістрів, є найпростішою реалізацією концепції адресного простору процесу, яка ґрунтується на тому, що кожний процес має власне відображення пам'яті.

Для організації захисту пам'яті в цій ситуації необхідно, щоб застосування користувача не могли змінювати значення базового і межового регістрів. Достатньо інструкції такої зміни зробити доступними тільки у привілейованому режимі процесора.

До переваг цього підходу належать простота, незначні вимоги до апаратного забезпечення (потрібні тільки два регістри), висока ефективність. Однак сьогодні його практично не використовують через низку недоліків, пов'язаних насамперед з тим, що адресний простір процесу все одно відображається на один неперервний блок фізичної пам'яті: незрозуміло, як динамічно розширювати адресний простір процесу; різні процеси не можуть спільно використовувати пам'ять; немає розподілу коду і даних.

За такого підходу для процесу виділяють тільки одну пару значень «базова адреса-межа». Природним розвитком цієї ідеї стало відображення адресного простору процесу за допомогою кількох діапазонів фізичної пам'яті, кожен з яких задають власною парою значень базової адреси і межі. Так виникла концепція сегментації пам'яті.

5. Сегментація пам'яті

Сегмент пам'яті це сукупність незалежних блоків змінної довжини. Сегментація пам'яті дає змогу зображати логічний адресний простір як сукупність сегментів. Кожний сегмент звичайно містить дані одного призначення, наприклад в одному може бути стек, в іншому – програмний код і т. д.

У кожного сегмента є ім'я і довжина. Логічна адреса складається з номера сегмента і зсуву всередині сегмента; з такими адресами працює прикладна програма. Компілятори часто створюють окремі сегменти для різних даних програми (сегмент коду, сегмент даних, сегмент стеку). Під час завантаження програми у пам'ять створюють таблицю дескрипторів сегментів процесу, кожний елемент якої відповідає одному сегменту і складається із базової адреси, значення межі та прав доступу.

Під час формування адреси її сегментна частина вказує на відповідний елемент таблиці дескрипторів сегментів процесу. Якщо зсув більший, ніж задане значення межі (або якщо права доступу процесу не відповідають правам, заданим для сегмента), то апаратне забезпечення генерує помилку. Коли ж усе гаразд, сума бази і зсуву в разі чистої сегментації дасть у результаті фізичну адресу в основній пам'яті. Якщо сегмент вивантажений на диск, спроба доступу до нього спричиняє його завантаження з диска в основну пам'ять. У підсумку кожному сегменту відповідає неперервний блок пам'яті такої самої довжини, що перебуває в довільному місці фізичної пам'яті або на диску. Загальний підхід до перетворення адреси у разі сегментації показаний на рис. 5.

Загальний вигляд пам'яті у випадку сегментації показаний на рис. 6.

Сегментацію застосовують доволі обмежено через фрагментацію і складність реалізації ефективного звільнення пам'яті та обміну із диском. Ширше використання отримав розподіл пам'яті на блоки фіксованої довжини – сторінкова організація пам'яті.

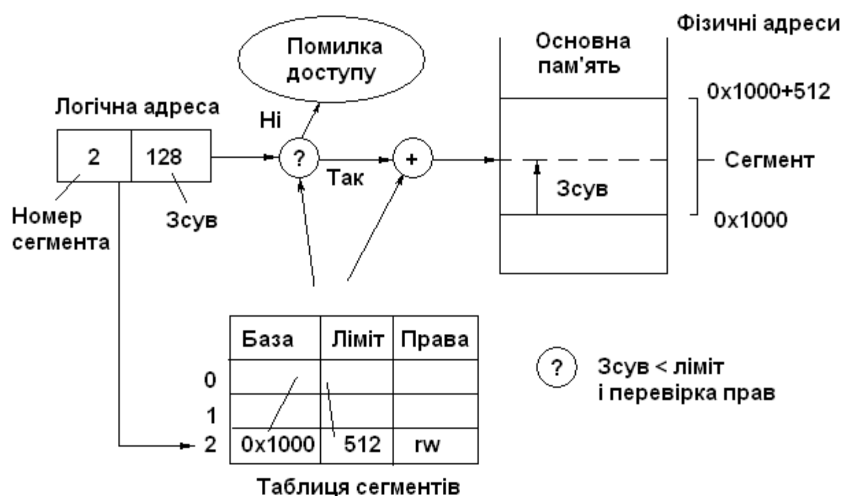

Рисунок 5 – Перетворення адреси при сегментації

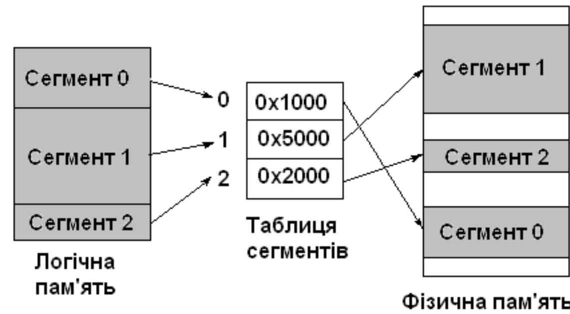

Рисунок 6 – Логічний і фізичний адресний простір при сегментації

6 Сторінкова організація пам’яті

До основних технологій реалізації віртуальної пам’яті крім сегментації належить *сторінкова організація пам’яті* (paging). *Сторінка* це блок пам’яті фіксованої довжини. Головна ідея сторінкової організації пам’яті – розподіл пам’яті сторінками. Ця технологія є найпоширенішим підходом до реалізації віртуальної пам’яті у сучасних ОС.

При сторінковій організації пам’яті логічну адресу називають також лінійною, або віртуальною, адресою. Такі адреси належать одній множині (наприклад, лінійною адресою може бути невід’ємне ціле число довжиною 32 біти).

Фізичну пам’ять розбивають на блоки фіксованої довжини – фрейми, або сторінкові блоки (frames). Логічну пам’ять, у свою чергу, розбивають на блоки такої самої довжини – сторінки (pages). Коли процес починає виконуватися, його сторінки завантажуються в доступні фрейми фізичної пам’яті з диска або іншого носія.

Сторінкова організація пам’яті повинна мати апаратну підтримку. Кожна адреса, яку генерує процесор, ділиться на дві частини: номер сторінки і зсув сторінки. Номер сторінки використовують як індекс у таблиці сторінок.

Таблиця сторінок – це структура даних, що містить набір елементів (page-table entries, PTE), кожен із яких містить інформацію про номер сторінки, номер відповідного їй фрейму фізичної пам’яті (або безпосередньо його базову адресу) та права доступу. Номер сторінки використовують для пошуку елемента в таблиці. Після його знаходження до базової адреси відповідного фрейму додають зсув сторінки, чим і визначають фізичну адресу (рис. 7).

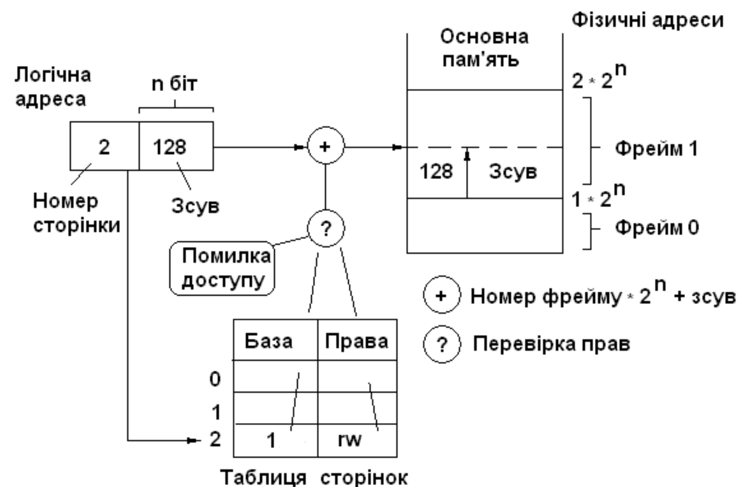

Рисунок 7 – Перетворення адреси при сторінковій організації пам’яті

Розмір сторінки є ступенем числа 2, у сучасних ОС використовують сторінки розміром від 2 до 8 Кбайт. У спеціальних режимах адресації можна працювати зі сторінками більшого розміру.

Для кожного процесу створюють його власну таблицю сторінок. Коли процес починає своє виконання, ОС розраховує його розмір у сторінках і кількість фреймів у фізичній пам'яті. Кожну сторінку завантажують у відповідний фрейм, після чого його номер записують у таблицю сторінок процесу.

Відображення логічної пам'яті для процесу відрізняється від реального стану фізичної пам'яті. На логічному рівні для процесу вся пам'ять зображується неперервним блоком і належить тільки цьому процесові, а фізично вона розосереджена по адресному простору мікросхеми пам'яті, чергуючись із пам'яттю інших процесів (рис. 8). Процес не може звернутися до пам'яті, адреса якої не вказана в його таблиці сторінок (так реалізований захист пам'яті).

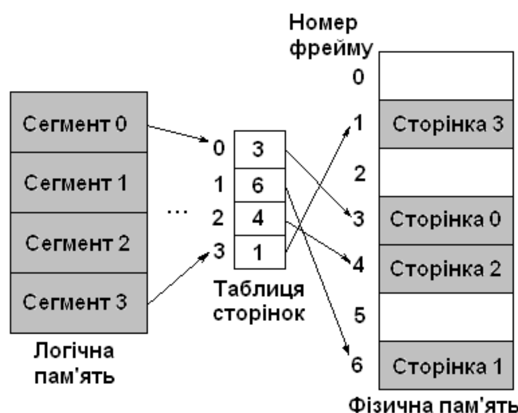

Рисунок 8 – Логічний і фізичний простір при сторінковій організації пам'яті

6.1 Багаторівневі таблиці сторінок

Щоб адресувати логічний адресний простір значного обсягу за допомогою однієї таблиці сторінок, її доводиться робити дуже великою. Наприклад, в архітектурі IA-32 за стандартного розміру сторінки 4 Кбайт (для адресації всередині такої сторінки потрібні 12 біт) на індекс у таблиці залишається 20 біт, що відповідає таблиці сторінок на 1 мільйон елементів.

Щоб уникнути таких великих таблиць, запропоновано технологію *багаторівневих таблиць сторінок*. Таблиці сторінок самі розбиваються на сторінки, інформацію про які зберігають в таблиці сторінок верхнього рівня. Кількість рівнів рідко перевищує 2, але може доходити й до 4. Коли є два рівні таблиць, логічну адресу розбивають на індекс у таблиці верхнього рівня, індекс у таблиці нижнього рівня і зсув.

Ця технологія має дві основні переваги. По-перше, таблиці сторінок стають менші за розміром, тому пошук у них можна робити швидше. По-друге, не всі таблиці сторінок мають перебувати в пам'яті у конкретний момент часу. Наприклад, якщо процес не використовує якийсь блок пам'яті, то вміст усіх сторінок нижнього рівня невикористовуваного блоку може бути тимчасово збережений на диску.

6.2 Реалізація таблиць сторінок в архітектурі IA_32

Архітектура IA-32 використовує дворівневу сторінкову організацію, починаючи з моделі Intel 80386.

Таблицю верхнього рівня називають *каталогом сторінок* (page directory), для кожної задачі повинен бути заданий окремий каталог сторінок, фізичну адресу якого зберігають у спеціальному керуючому регістрі `cr3` і куди він автоматично завантажується апаратним

забезпеченням при перемиканні контексту. Таблицю нижнього рівня називають просто *таблицею сторінок* (page table).

Лінійна адреса поділяється на три поля:

• *каталог* (Directory) — визначає елемент каталогу сторінок, що вказує на потрібну таблицю сторінок;

• *таблиця* (Table) — визначає елемент таблиці сторінок, що вказує на потрібний фрейм пам'яті;

• *зсув* (Offset) — визначає зсув у межах фрейму, що у поєднанні з адресою фрейму формує фізичну адресу.

Розмір полів каталогу і таблиці становить 10 біт, що дає таблиці сторінок, які містять 1024 елементи, розмір поля зсуву – 12 біт, що дає сторінки і фрейми розміром 4 КБайт. Одна таблиця сторінок нижнього рівня адресує 4 МБайт пам'яті (1 Мбайт фреймів), а весь каталог сторінок – 4 ГБайт.

Елементи таблиць сторінок всіх рівнів мають однакову структуру. Основні поля елемента таблиці:

• *прапор присутності* (Present), дорівнює одиниці, якщо сторінка перебуває у фізичній пам'яті (їй відповідає фрейм); рівність цього прапорця нулю означає, що сторінки у фізичній пам'яті немає, при цьому ОС може використати інші поля елемента для своїх цілей;

• *20 найбільш значущих бітів*, які задають початкову адресу фрейму, кратну 4 Кбайт (може бути задано 1 Мбайт різних початкових адрес);

• *прапор доступу* (Accessed), який задають рівним одиниці під час кожного звертання пристрою сторінкової підтримки до відповідного фрейму;

• *прапор зміни* (Dirty), який задають рівним одиниці під час кожної операції записування у відповідний фрейм;

• *прапор читання-записування* (Read/Write), що задає права доступу до цієї сторінки або таблиці сторінок (для читання і для записування або тільки для читання);

• *прапор привілейованого режиму* (User/Supervisor), який визначає режим процесора, необхідний для доступу до сторінки. Якщо цей прапор дорівнює нулю, сторінка може бути адресована тільки із привілейованого режиму, якщо одиниці – доступна також і з режиму користувача;

Прапори присутності, доступу і зміни може використовувати ОС для організації віртуальної пам'яті.

7 Асоціативна пам'ять

Під час реалізації таблиць сторінок для отримання доступу до байта фізичної пам'яті доводиться звертатися до пам'яті кілька разів. У разі використання дворівневих сторінок потрібні *три* операції доступу: до каталогу сторінок, до таблиці сторінок і безпосередньо за адресою цього байта, а для трирівневих таблиць – чотири операції. Це сповільнює доступ до пам'яті та знижує загальну продуктивність системи.

Як уже зазначалося, правило «дев'яносто до десяти» свідчить, що більша частина звертань до пам'яті процесу належить до малої підмножини його сторінок, причому склад цієї підмножини змінюється досить повільно. Засобом підвищення продуктивності у разі сторінкової організації пам'яті є кешування адрес фреймів пам'яті, що відповідають цій підмножині сторінок.

Для розв'язання цієї проблеми було запропоновано технологію *асоціативної пам'яті* або *кешу трансляції*, (translation look-aside buffers, TLB). У швидкодіючій пам'яті (швидшій, ніж основна пам'ять) створюють набір із кількох елементів (різні архітектури відводять під асоціативну пам'ять від 8 до 2048 елементів, в архітектурі IA-32 таких елементів до Pentium-4 було 32, починаючи з Pentium-4 – 128). Кожний елемент кешу трансляції відповідає одному елементу таблиці сторінок.

Тепер під час генерування фізичної адреси спочатку відбувається пошук відповідного елементу таблиці в кеші (в IA-32 – за полем каталогу, полем таблиці та зсуву), і якщо він знайдений, стає доступною адреса відповідного фрейму, що негайно можна використати для звертання до пам'яті. Якщо ж у кеші відповідного елементу немає, то доступ до пам'яті здійснюють через таблицю сторінок, а після цього елемент таблиці сторінок зберігають в кеші замість найстарішого елемента (рис. 9).

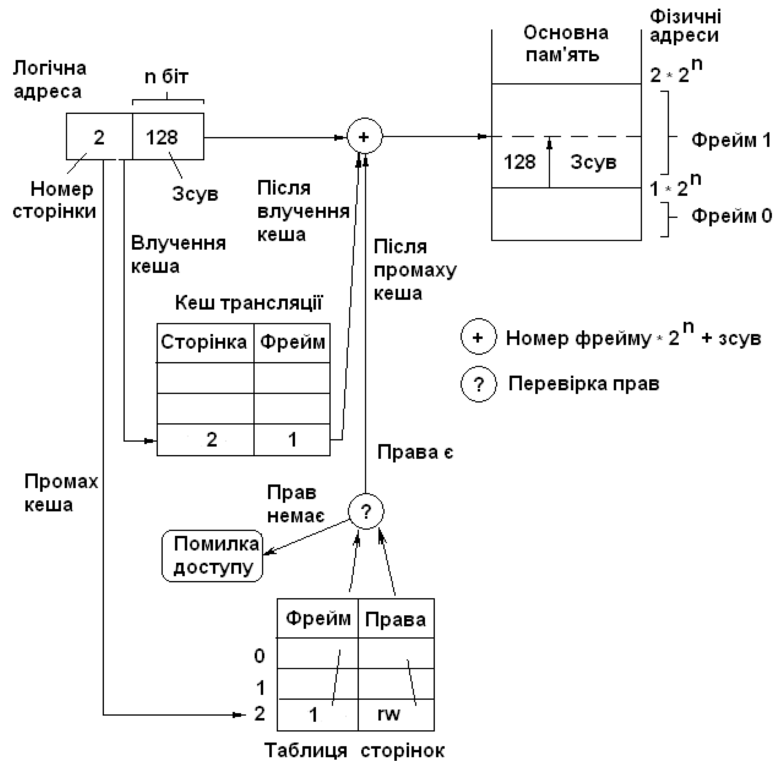

Рисунок 9 – Принцип роботи кешу трансляції

На жаль, у разі перемикання контексту в архітектурі IA-32 необхідно очистити весь кеш, оскільки в кожного процесу є своя таблиця сторінок, і ті ж самі номери сторінок для різних процесів можуть відповідати різним фреймам у фізичній пам'яті. Очищення кешу трансляції є дуже повільною операцією, якої треба всіляко уникати.

Важливою характеристикою кешу трансляції є відсоток влучень, тобто відсоток випадків, коли необхідний елемент таблиці сторінок перебуває в кеші і не потребує доступу до пам'яті. Відомо, що при 32 елементах забезпечується 98% влучень. Зазначимо також, що за такого відсотку влучень зниження продуктивності у разі використання дворівневих таблиць сторінок порівняно з однорівневими становить 28%, однак переваги, отримувані під час розподілу пам'яті, роблять таке зниження допустимим.

8. Сторінково-сегмента організація пам'яті

Оскільки сегменти мають змінну довжину і керувати ними складніше, чиста сегментація зазвичай не настільки ефективна, як сторінкова організація. З іншого боку, є цінною сама можливість використати сегменти як блоки пам'яті різного призначення змінної довжини.

Для того щоб об'єднати переваги обох підходів, у деяких апаратних архітектурах (зокрема, в IA-32) використовують комбінацію сегментної та сторінкової організації пам'яті. За такої організації перетворення логічної адреси у фізичну відбувається за три етапи.

1. У програмі задають логічну адресу із використанням сегмента і зсуву.

2. Логічну адресу перетворюють у лінійну (віртуальну) адресу за правилами, заданими для сегментації.

3. Віртуальну адресу перетворюють у фізичну за правилами, заданими для сторінкової організації.

Таку архітектуру називають *сторінково-сегментною організацією пам'яті*.

Особливості реалізації описаних трьох етапів перетворення адреси в архітектурі IA-32.

1. Машинна мова архітектури IA-32 оперує логічними адресами. Логічна адреса складається із селектора на дескриптор сегменту і зсуву.

2. Лінійна або віртуальна адреса – це ціле число без знаку завдовжки 32 біти. За його допомогою можна дістати доступ до 4 ГБайт комірок пам'яті. Перетворення логічної адреси в лінійну відбувається всередині *пристрою сегментації* (segmentation unit) за правилами перетворення адреси на базі сегментації.

3. Фізичну адресу використовують для адресації комірок пам'яті в мікросхемах пам'яті. Її теж зображають 32-бітовим цілим числом без знаку. Перетворення лінійної адреси у фізичну відбувається всередині пристрою сторінкової підтримки (paging unit) за правилами для сторінкової організації пам'яті (лінійну адресу розділяють апаратурою на адресу сторінки і сторінковий зсув, а потім перетворюють у фізичну адресу із використанням таблиць сторінок, кешу трансляції тощо).

Формування адреси для сторінково-сегментної організації пам'яті показано на рис. 10.

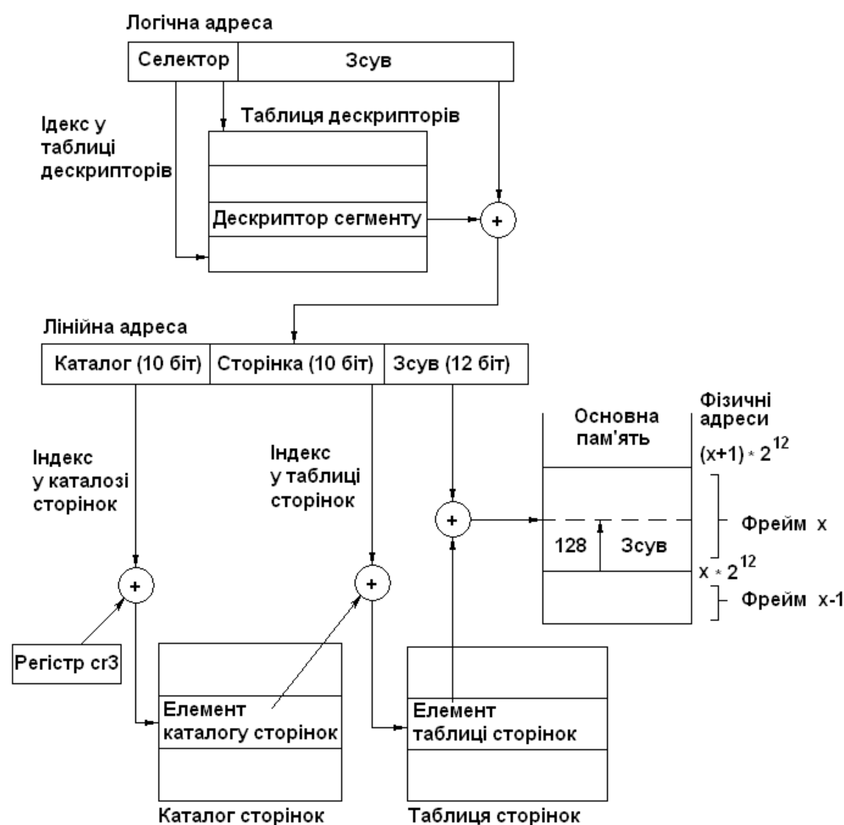

Рисунок 10 – Сторінково-сегментна організація пам'яті

Висновки.

• Завдяки віртуальній пам'яті фізична пам'ять адресного простору процесу може бути фрагментованою, оскільки основний обсяг пам'яті, яку займає процес, більшу частину часу залишається вільним.

• Найважливішими поняттями концепції віртуальної пам'яті є логічна і фізична адресація пам'яті.

• Під час реалізації віртуальної пам'яті необхідно забезпечити захист пам'яті, переміщення процесів у пам'яті та спільне використання пам'яті кількома процесами.

• Сегментація пам’яті дає змогу зображати логічний адресний простір як сукупність незалежних блоків змінної довжини, які називають сегментами.

• При сторінковій організації фізичну пам’ять розбивають на блоки фіксованої довжини – фрейми, або сторінкові блоки (frames), а логічну пам’ять, у свою чергу, розбивають на блоки такої самої довжини – сторінки (pages).

• Щоб адресувати логічний адресний простір значного обсягу використовують технологію багаторівневих таблиць сторінок.

• Засобом підвищення продуктивності у випадку сторінкової організації пам’яті є кешування адрес фреймів пам’яті, що відповідають цій підмножині сторінок.

• У деяких апаратних архітектурах (зокрема, в IA-32) використовують комбінацію сегментної та сторінкової організації пам’яті.

Література.

1. *Гордеев В.В., Молчанов А.Ю.* Системное программное обеспечение. – СПб.: Питер, 2001. – 736.

2. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.

Запитання.

1. Що таке віртуальна пам'ять,
2. Що таке зовнішня і внутрішня фрагментація пам’яті?
3. Що таке фізична і логічна адреса?
4. Що таке сегментація пам’яті?
5. Що таке сторінкова організація пам’яті і таблиця сторінок?
6. Що таке асоціативна пам’ять?
7. Особливості сторінково-сегментної організації пам’яті.

5. ОРГАНІЗАЦІЯ ФАЙЛОВИХ СИСТЕМ

Мета. Вивчення логічної і фізичної організації файлових систем.

Вступ. Файлові системи можна розглядати на двох рівнях: логічному і фізичному. Логічний рівень визначає відображення файлової системи, призначене для прикладних програм і користувачів, фізичний – особливості розташування структур даних системи на диску й алгоритми, які використовують під час доступу до інформації.

План.
1 Основні визначення
2 Логічна організація файлових систем
2.1 Поняття файла і файлової системи
2.2 Організація інформації у файловій системі
2.3 Зв'язки
2.4. Операції над файлами і каталогами
2 Фізична організація файлових систем
2.1 Принцип дії жорсткого диска
2.2 Розділи диска
2.3 Стандарт GPT
2.4 Неперервне розміщення файлів
2.5 Розміщення файлів зв'язаними списками
2.6 Індексоване розміщення файлів
3 Файлова система NTFS
4 Файлові система UNIX

1 Основні визначення

Файлова система (ФС) – це частина ОС, призначення якої полягає в тому, щоб забезпечити користувачу зручний інтерфейс для роботи з даними, що зберігаються на диску, а також сумісне використання файлів декількома користувачами і процесами.

У широкому сенсі поняття «файлова система» охоплює:

− сукупність всіх файлів на диску;

− набори структур даних, використовуваних для керування файлами, такі, наприклад, як каталоги файлів, дескриптори файлів, таблиці розподілу вільного і зайнятого простору на диску;

− комплекс системних програмних засобів, що реалізовують керування файлами, зокрема: створення, знищення, зчитування, запис, іменування, пошук та інші операції над файлами.

Іменування файлів. Правила іменування файлів залежать від ОС:

− у багатьох ОС підтримуються імена з двох частин (імені та розширення), наприклад `prog.c` (файл, що містить текст програми мовою С) або `autoexec.bat` (файл, що містить команди інтерпретатора командної мови);

− тип розширення файла дозволяє ОС організувати роботу з ним для різних прикладних програм за наперед узгодженими правилами;

− звичайно ОС накладають деякі обмеження на використовувані в імені символи і на довжину імені файла;

− стандарт POSIX встановлює зручні для користувача довгі імена (до 255 символів).

Типи файлів. Файли розрізняють за типами:

– звичайний;
  • текстовий;
  • двійковий;

– спеціальний;
– каталог.

2 Логічна організація файлових систем

Види логічної організації ФС показано на рис. 1.

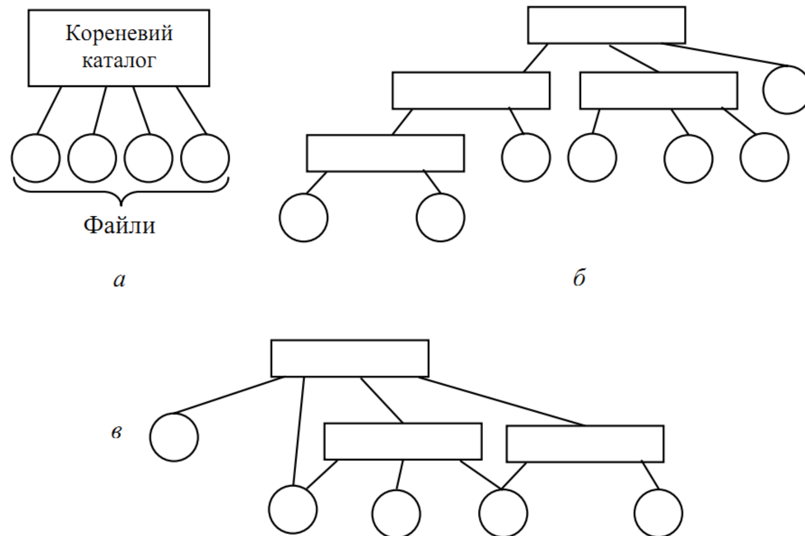

Рисунок 1 – Логічна організація ФС:
а – однорівнева; б – ієрархічна; в – ієрархічна мережа

2.1 Поняття файла і файлової системи

Файл (file) – це набір даних, до якого можна звертатися за іменем. Файли організовані у файлові системи. З погляду користувача файл є мінімальним обсягом даних файлової системи, з яким можна працювати незалежно. Наприклад, користувач не може зберегти дані на зовнішньому носії без звернення до файла.

Файли поділяються на файли з прямим і послідовним доступом. Файли з *прямим доступом* дають змогу вільно переходити до будь-якої позиції у файлі, використовуючи для цього поняття вказівника поточної позиції (seek pointer), який може переміщатися у будь-якому напрямку за допомогою відповідних системних викликів.

Файли із *послідовним доступом* можуть бути зчитані тільки послідовно, із початку в кінець. Сучасні ОС звичайно розглядають усі файли як файли із прямим доступом.

Доступ до файлів здійснюється за їх іменами. Стандартним значенням максимальної довжини імені файла є 255 символів.

Файлова система (file system) – це підсистема ОС, що підтримує організований набір файлів, здебільшого у конкретній ділянці дискового простору (логічну структуру); низькорівневі структури даних, використовувані для організації цього простору у вигляді набору файлів (фізичну структуру); програмний інтерфейс файлової системи (набір системних викликів, що реалізують операції над файлами).

Файлова система надає прикладним програмам абстракцію файла. Прикладні програми не мають інформації про те, як організовані дані файла, як знаходять відповідність між ім’ям файла і його даними, як пересилають дані із диска у пам’ять тощо — усі ці операції забезпечує файлова система.

Важливо зазначити, що файлові системи можуть надавати інтерфейс доступу не тільки до диска, але й до інших пристроїв. Є навіть файлові системи, які не зберігають інформацію, а генерують її динамічно за запитом. Втім, для прикладних програм усі такі системи мають однаковий вигляд.

Кожний файл має набір характеристик – *атрибутів*. Набір атрибутів змінюється залежно від файлової системи. Найпоширеніші атрибути файла:

• ім’я файла;

• тип файла, який звичайно задають для спеціальних файлів (каталогів, зв’язків тощо).

• розмір файла (поточний і максимальний);

• атрибути безпеки, що визначають права доступу до цього файла;

• часові атрибути, до яких належать час створення останньої модифікації та останнього використання файла;

Інформацію про атрибути файла також зберігають на диску. Особливості її зберігання залежать від фізичної організації файлової системи.

2.2 Організація інформації у файловій системі

Розділ (partition) – частина фізичного дискового простору, що призначена для розміщення на ній структури однієї файлової системи і з логічної точки зору розглядається як єдине ціле. Розділ – це логічний пристрій, що з погляду ОС функціонує як окремий диск. Такий пристрій може відповідати всьому фізичному диска (у цьому разі кажуть, що диск містить один розділ); найчастіше він відповідає частині диска (таку частину називають ще фізичним розділом); буває й так, що подібні логічні пристрої поєднують кілька фізичних розділів, що перебувають, можливо, на різних дисках (такі пристрої ще називають *логічними томами* – logical volumes).

Кожний розділ може мати свою файлову систему (і, можливо, використовуватися різними ОС). Для поділу дискового простору на розділи використовують спеціальну утиліту, яку часто називають `fdisk`. Для створення файлової системи потрібно виконати операцію високорівневого форматування диска.

У деяких ОС під *томом* (volume) розуміють розділ із встановленою на ньому файловою системою. Реалізація розділів дає змогу відокремити логічне відображення дискового простору від фізичного і підвищує гнучкість використання файлових систем.

Каталог – це об’єкт (найчастіше реалізований як спеціальний файл), що містить інформацію про набір файлів. Про такі файли кажуть, що вони містяться в каталозі. Файли заносяться в каталоги користувачами на підставі їхніх власних критеріїв, деякі каталоги можуть містити дані, потрібні операційній системі, або її програмний код.

Каталог можна уявити собі як символьну таблицю, що реалізує відображення імен файлів у елементи каталогу (звичайно в таких елементах зберігають низькорівневу інформацію про файли).

Базовою ідеєю організації даних за допомогою каталогів є те, що вони можуть містити інші каталоги. Вкладені каталоги називають *підкаталогами* (subdirectories). Таким чином формується дерево каталогів. Перший каталог, створений у файловій системі (корінь дерева каталогів), називають *кореневим каталогом* (root directory).

Для файла, розміщеного всередині каталогу недостатньо його імені для однозначного визначення, де він перебуває, – в іншому каталозі може бути файл із тим самим ім’ям. Тепер для визначення місцезнаходження файла потрібно додавати до його імені список каталогів, де він перебуває. Такий список називається *шляхом* (path). Каталоги у шляху перераховують зліва направо – від меншої глибини вкладеності до більшої. Роздільник каталогів у шляху відрізняється для різних систем: в UNIX прийнято використовувати пряму похилу “/”, а у Windows-системах – зворотну похилу “\”.

Є два шляхи до файла: абсолютний і відносний. *Абсолютний* (або повний) однозначно визначає місце розташування файла. Такий шлях обов’язково має містити кореневий каталог. Приклад абсолютного шляху для UNIX-систем: `/usr/local/bin/myfile`. Якщо застосунок використовує тільки абсолютні шляхи, йому звичайно бракує гнучкості. Наприклад, у разі перенесення в інший каталог потрібно буде вручну відредагувати всі шляхи, замінивши їх новими.

Відносний – шлях, відраховується від деякого місця в ієрархії каталогів. Щоб його організувати, потрібно визначитися із точкою відліку, для чого використовують поняття *поточного каталогу*. Такий каталог задають для кожного процесу, і він може бути змінений у будь-який момент командою `cd` або системним викликом `chdir()`. Відносний шлях може відраховуватися від поточного каталогу і звичайно кореневий каталог не включає. Прикладом відносного шляху до файла `/usr/local/bin/myfile` (за умови, що поточним каталогом є `/usr/local`) буде `bin/myfile`, а в ситуації, коли поточним є каталог файла (`/usr/local/bin`), відносним шляхом буде просто ім'я файла: `myfile`.

Для спрощення побудови відносного шляху кожний каталог містить два спеціальні елементи:

• ".", що посилається на поточний каталог;

• "..", що посилається на каталог рівнем вище.

З урахуванням цих елементів можуть бути задані такі відносні шляхи, як `../../bin/myfile` (за умови, що поточний каталог – `/usr/local/lib/mylib`) або `./myfile` (вказує на елемент у поточному каталозі).

У файловій системі UNIX використовується єдине дерево каталогів ОС. Стандартну організацію каталогів UNIX зображують у вигляді дерева з одним коренем – кореневим каталогом, який позначають "/". Файлову систему, у якій перебуває кореневий каталог, називають завантажувальною або кореневою. У більшості реалізацій вона має містити файл із ядром ОС.

Додаткові файлові системи об'єднуються із кореневою за допомогою операції *монтування* (mount). Під час монтування вибраний каталог однієї файлової системи стає кореневим каталогом іншої. Каталог, призначений для монтування файлової системи, називають *точкою монтування* (mount point). Весь вміст файлової системи, приєднаної за допомогою монтування, виглядає для користувачів системи як набір підкаталогів точки монтування.

2.3 Зв'язки

Структура каталогів файлової системи не завжди є деревом. Багато файлових систем дає змогу задавати кілька імен для одного й того самого файла. Такі імена називають зв'язками (links). Розрізняють жорсткі та символьні зв'язки.

Ім'я файла не завжди однозначно пов'язане з його даними. За підтримки *жорстких зв'язків* (hard links) для файла допускається декілька імен. Усі жорсткі зв'язки визначають одні й ті самі дані на диску, для користувача вони не відрізняються: не можна визначити, які з них були створені раніше, а які – пізніше.

Для створення жорстких зв'язків у POSIX призначений системний виклик `link()`. Першим параметром він приймає ім'я вихідного файла, другим – ім'я жорсткого зв'язку, який буде створений:

```
#include <unistd.h> // для стандартних файлових операцій POSIX
link ("myfile.txt", "myfile-hardlink.txt"):
```

Зазначимо, що стандартні засоби вилучення даних за наявності жорстких зв'язків працюватимуть саме з ними, а не безпосередньо із файлами. Замість системного виклику вилучення файла використовують виклик вилучення зв'язку (`unlink()`), що вилучатиме один жорсткий зв'язок для заданого файла. Якщо після цього зв'язків у файлі більше не залишається, його дані також вилучаються.

```
// вилучити файл, якщо в нього був один жорсткий зв'язок
unlink("myfile.txt");
```

Символьний зв'язок (посилання) - зв'язок, фізично відокремлений від даних, на які вказує. Фактично, це спеціальний файл, що містить ім'я файла, на який вказує. Наведемо властивості символьних зв'язків.

• Через такий зв'язок здійснюють доступ до початкового файла.

• При вилученні зв'язку, вихідний файл не зникне.

• Якщо початковий файл перемістити або вилучити, зв'язок розірветься, і доступ через нього стане неможливий, якщо файл потім поновити на тому самому місці, зв'язком знову можна користуватися.

• Символьні зв'язки можуть вказувати на каталоги і файли, що перебувають в інших файлових системах (в іншому розділі жорсткого диска). Наприклад, якщо створити в поточному каталозі зв'язок `system-docs`, що вказує на каталог `/usr/doc`, то перехід у каталог `system-docs` призведе до переходу в каталог `/usr/doc`.

Для задання символьного зв'язку у POSIX визначено системний виклик `symlink()`, параметри якого аналогічні до параметрів `link()`:

```
symlink(”myfile.txt”, ”myfile-symlink.txt”);
```

Для отримання шляху до файла або каталогу, на який вказує символьний зв'язок, використовують системний виклик `readlink()`:

```
// PATH_MAX - константа, що задає максимальну довжину шляху
char filepath[PATH_MAX+l];
readlink(”myfile-symlink.txt”, filepath, sizeof(filepath));
// у filepath буде шлях до myfile.txt
```

2.4 Операції над файлами і каталогами

ОС підтримує наступні файлові операції для використання у прикладних програмах.

• *Відкриття файла*. Після відкриття файла процес може з ним працювати (наприклад, робити читання і записування). Відкриття файла звичайно передбачає завантаження в оперативну пам'ять спеціальної структури даних – дескриптора файла, який визначає його атрибути та місце розташування на диску.

Наступні виклики використовуватимуть цю структуру для доступу до файла.

• *Закриття файла.* Після завершення роботи із файлом його треба закрити. При цьому структуру даних, створену під час його відкриття, вилучають із пам'яті. Усі дотепер не збережені зміни записують на диск.

• *Створення файла*. Ця операція спричиняє створення на диску нового файла нульової довжини. Після створення файл автоматично відкривають.

• *Вилучення файла*. Ця операція спричиняє вилучення файла і вивільнення зайнятого ним дискового простору. Вона звичайно недопустима для відкритих файлів.

• *Читання з файла*. Ця операція зводиться до пересилання певної кількості байтів із файла, починаючи із поточної позиції, у заздалегідь виділений для цього буфер пам'яті режиму користувача.

• Записування у файл. Здійснюється із поточної позиції, дані записуються у файл із заздалегідь виділеного буфера. Якщо на цій позиції вже є дані, вони будуть перезаписані. Ця операція може змінити розмір файла.

• *Переміщення вказівника поточної позиції.* Перед операціями читання і записування слід визначити, де у файлі перебувають потрібні дані або куди треба їх записати, задавши за допомогою цієї операції поточну позицію у файлі. Зазначимо, що якщо перемістити вказівник файла за його кінець, а потім виконати операцію записування, довжина файла збільшиться.

• *Отримання і встановлення атрибутів файла.* Ці дві операції дають змогу зчитувати поточні значення всіх або деяких атрибутів файла або задавати для них нові значення.

Для відкриття файла використовують системний виклик `open()`, першим параметром якого є шлях до файла.

```
#include <fcntl.h>
int open(const char *pathname, int flags [, mode_t mode]);
```

Виклик `open()` повертає цілочислове значення – *файловий дескриптор*. Його слід використовувати в усіх викликах, яким потрібен відкритий файл. У разі помилки цей виклик поверне -1, а значення змінної `errno` відповідатиме коду помилки.

Параметр `flags` може набувати наступні значення (їх можна об’єднувати за допомогою побітового “або”):

• `O_RDONLY`, `O_WRONLY`, `O_RDWR` – відкриття файла, відповідно, тільки для читання, тільки для записування або для читання і записування (має бути задане одне із цих трьох значень, наведені нижче не обов’язкові);

• `O_CREAT` – якщо файл із таким ім’ям відсутній, його буде створено, якщо файл є і увімкнуто прапорець `O_EXCL`, буде повернено помилку;

• `O_TRUNC` – якщо файл відкривають для записування, його довжину приймають рівною нулю;

• `O_NONBLOCK` – задає неблокувальне введення-виведення.

Приклад використання системного виклику:

```
// відкриття файла для записування
int fdl = open(”./myfile.txt”, O_WRONLY|O_CREAT|O_TRUNC, 0644);
// відкриття файла для читання, помилка, якщо файла немає
int fdl = open(”./myfile.txt”, O_RDONLY);
```

Файл закривають за допомогою системного виклику `close()`, що приймає файловий дескриптор:

```
close(fdl);
```

Для читання даних із відкритого файла використовують системний виклик `read()`:

```
ssize_t read(int fdl, void *buf. size_t count);
```

Внаслідок цього виклику буде прочитано `count` байтів із файла, заданого відкритим дескриптором `fdl`, у пам’ять, на яку вказує `buf` (ця пам’ять виділяється заздалегідь). Виклик `read()` повертає реальний обсяг прочитаних даних (тип `ssize_t`) є цілочисловим). Вказівник позиції у файлі пересувають за зчитані дані.

```
char buf[100];
// читають 100 байт з файла в buf
int bytes_read = read(fdl, buf, sizeof(buf));
```

Коли потрібна кількість даних у конкретний момент відсутня (наприклад, `fdl` пов’язаний із мережним з’єднанням, яким ще не прийшли дані), поведінка цього виклику залежить від значення прапора `O_NONBLOCK` під час виклику `open()`. У разі блокувального виклику (`O_NONBLOCK` не увімкнуто) він призупинить поточний потік до тих пір, поки дані не з’являться, а в разі неблокувального (прапорець `O_NONBLOCK` увімкнуто) – зчитає всі доступні дані й завершиться, призупинення потоку не відбудеться.

Для записування даних у відкритий файл через файловий дескриптор використовують системний виклик `write()`:

```
ssize_t write(int fdl, const void *buf, size_t count);
```

Внаслідок цього виклику буде записано `count` байтів у файл через дескриптор `fdl` із пам’яті, на яку вказує `buf`. Виклик `write()` повертає обсяг записаних даних.

```
int fdl, bytes_written;
fdl = open(”./myfile.txt”, O_RDWR|O_CREAT, 00644);
bytes_written = write(fdl, ”hello”, sizeof(”hello”);
```

Кожному відкритому файла відповідає *вказівник позиції* (зміщення) усередині файла. Вказівник позиції *можна пересунути* за допомогою системного виклику `lseek()`:

```
off_t lseek(int fdl, off_t offset, int whence);
```

Параметр `offset` задає величину переміщення вказівника. Режим переміщення задають параметром `whence`, який може набувати значень `SEEK_SET` (абсолютне переміщення від початку файла), `SEEK_CUR` (відносне переміщення від поточного місця вказівника позиції) і `SEEK_END` (переміщення від кінця файла).

```
// переміщення вказівника позиції на 100 байт від поточного місця
lseek(outfile, 100, SEEK_CUR);
write(outfile, ”hello”, sizeof(”hello”)); // записування у файл
```

Коли вказівник поточної позиції перед операцією записування опиняється за кінцем файла, він внаслідок записування автоматично розширюється до потрібної довжини. На цьому ґрунтується ефективний спосіб створення файлів необхідного розміру:

```
int fdl = open(”file”, O_RDWR|O_CREAT|O_TRUNC, 0644);// створення файла
1seek(fdl, needed_size,SEEK_SET);  // розширення до потрібного розміру
write(fdl, ””, 1); // записування нульового байта
```

Для отримання інформації про *атрибути файла* (тобто про вміст його індексного дескриптора) використовують системний виклик `stat()`.

```
finclude <sys/stat.h>
int stat(const char *path, struct stat *attrs);
```

Першим параметром є шлях до файла, другим – структура, у яку записуватимуться атрибути внаслідок виклику. Деякі поля цієї структури (всі цілочислові) наведено нижче:

• `st_mode` – тип і режим файла (бітова маска прапорів, зокрема прапор `S_IFDIR` встановлюють для каталогів);

• `st_nlink` – кількість жорстких зв’язків;

• `st_size` – розмір файла у байтах;

• `st_atime, st_mtime, st_ctime` – час останнього доступу, модифікації та зміни атрибутів (у секундах з 1 січня 1970 року).

Приклад відображення інформації про атрибути файла:

```
struct stat attrs;
stat(”myfile”, &attrs);
if (attrs.st_mode & S_IFDIR)
  printf(”myfile в каталогом\n”):
else printf(”розмір файла: %d\n”,attrs.st_size);
```

Для отримання такої самої інформації з дескриптора відкритого файла використовують виклик `fstat()`:

```
int fstat(int fdl, struct stat *attrs);
```

ОС підтримує наступні базові операції з каталогами:

• *Створення нового каталогу*. Ця операція створює новий каталог. Він звичайно порожній, деякі реалізації автоматично додають у нього елементи “.” і “..”.

• *Вилучення каталогу***.** На рівні системного виклику ця операція дозволена тільки для порожніх каталогів.

• *Відкриття і закриття каталогу*. Каталог, подібно до звичайного файла, має бути відкритий перед використанням і закритий після використання. Деякі операції, пов’язані із доступом до елементів, допустимі тільки для відкритих каталогів.

• *Читання елемента каталогу*. Ця операція зчитує один елемент каталогу і переміщує поточну позицію на наступний елемент. Використовуючи читання елемента каталогу в циклі, можна обійти весь каталог.

• Перехід у початок каталогу. Ця операція переміщує поточну позицію до першого елемента каталогу.

Робота з каталогами POSIX.

Для *створення каталогу* використовують виклик `mkdir()`, який приймає як параметр шлях до каталогу і режим.

```
if (mkdir(”./newdir”, 0644) == -1)
  printf(”помилка під час створення каталогу\n”);
```

Вилучення порожнього каталогу за його іменем відбувається за допомогою виклику `rmdir()`:

```
if (rmdir(”./dir”) == -1)
  printf(”помилка вилучення каталогу\n”);
```

Відкривають каталог викликом `opendir()`, що приймає як параметр ім’я каталогу:

```
DIR *opendir(const char *dirname);
```

Під час виконання `opendir()` ініціалізується внутрішній вказівник поточного елемента каталогу. Цей виклик повертає дескриптор каталогу – вказівник на структуру типу `DIR`, що буде використана під час обходу каталогу. При помилці повертається `NULL`.

Для *читання елемента каталогу* і переміщення внутрішнього вказівника поточного елемента використовують виклик `readdir()`:

```
struct dirent *readdir(DIR *dirp);
```

Цей виклик повертає вказівник на структуру `dirent`, що описує елемент каталогу (із полем `d_name`, яке містить ім'я елемента) або `NULL`, якщо елементів більше немає.

Після закінчення пошуку потрібно закрити каталог за допомогою виклику `closedir()`. Якщо необхідно перейти до першого елемента каталогу без його закриття, використовують виклик `rewinddir()`. Обидва ці виклики приймають як параметр дескриптор каталогу.

Приклад коду обходу каталогу в POSIX.

```
DIR *dirp; struct dirent *dp;
dirp = opendir(”,/dir”);
if (! dirp) { printf(”помилка відкриття каталогу\n”); exit(-l): }
while (dp = readdir(dirp)) {
  printf (”%s\n”, dp->d_name); // відображення імені елемента
}
closedir(dirp);
```

2 Фізична організація файлових систем

2.1 Принцип дії жорсткого диска

Накопичувачі на жорстких магнітних дисках (НЖМД) (далі – диски) складаються з набору дискових пластин, які покриті магнітним матеріалом і обертаються двигуном із високою швидкістю, рис. 2. Кожній пластині відповідають дві *головки* (heads), одна зчитує інформацію зверху, інша – знизу. Головки прикріплені до спеціального *дискового маніпулятора* (disk arm). Маніпулятор може переміщатися по радіусу диска – від центра до зовнішнього краю і назад, таким чином відбувається позиціонування головок.

Головки зчитують інформацію із *доріжок* (tracks), які мають вигляд концентричних кіл. Мінімальна кількість доріжок на поверхні пластини в сучасних дисках – 700, максимальна – більше 20000. Сукупність усіх доріжок одного радіуса на всіх поверхнях пластин називають *циліндром* (cylinder). Циліндри, як і доріжки нумеруються від краю диска до середини. Сектори нумеруються від позначки. Фізична нумерація головок починаються з 0, циліндрів – з 0, а секторів – з 1.

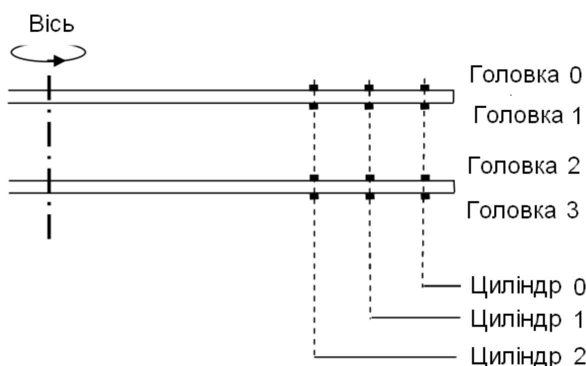

Рисунок 2 – Фізична структура диска

Кожну доріжку під час низькорівневого форматування розбивають на *сектори* (sectors). Кількість секторів для всіх доріжок однакова (у діапазоні від 16 до 1600). При записуванні інформації на диски спочатку заповнюються всі доріжки циліндра 0, потім циліндра 1 і т.д.

Кожен сектор складається з *поля даних і поля службової інформації,* що обмежує й ідентифікує його. Розмір сектора (точніше – ємність поля даних) встановлюється

контролером чи драйвером. У більшості сучасних ОС розмір сектора вибирається рівним 512 байт або кратним ступеню числа 2. Існує дві системи нумерації секторів диска – фізична (абсолютна) і логічна (відносна). Фізична адреса сектора на диску визначається за допомогою трьох ”координат”, тобто подається тріадою [C-H-S], де C – номер циліндра (доріжки на поверхні диска), H – номер магнітної головки, а S – номер сектора на доріжці.

ОС розподіляє дисковий простір не секторами, а спеціальними одиницями розміщення – *кластерами* (clusters) або *дисковими блоками* (disk blocks, термін «дисковий блок» більш розповсюджений в UNIX-системах). Визначення розміру кластера і розміщення інформації, необхідної для функціонування файлової системи, відбувається під час *високорівневого форматування розділу* (partition). Саме таке форматування створює файлову систему в розділі.

2.2 Розділи диска

Розділи диска можуть бути двох типів – *первинний* (primary) і *розширений* (extended). Максимальне число первинних розділів дорівнює *чотирьом*. При цьому на диску обов’язково повинний бути принаймні один первинний розділ. Якщо первинних розділів декілька, то тільки один з них може бути *завантажувальним* (bootable) або *активним* (active). Усередині розділу розташовані структури даних файлової системи.

Відповідно до специфікацій на одному жорсткому диска замість одного з первинних може бути один розширений розділ, який можна розділити до 24 *логічних дисків* (logical).

Один диск може містити декілька різних ОС, розміщених у різних розділах диска. В ході початкового завантаження можна вказати розділ диска, з якого повинна завантажуватися ОС.

Головний завантажувальний запис і таблиця розділів диска. З активного розділу завантажується програма завантаження завантажувача з якого-небудь іншого розділу, і вже за його допомогою завантажується ОС. Оскільки до завантаження ОС система керування файлами працювати не може, то для адресації завантажувачів ОС використовуються абсолютні адреси у форматі Cylinder-Header-Sector [C-H-S]. CHS може адресувати тільки розділи розміром до 7,8 Гб. Для розділів більших розділів ОС використовує логічну LBA адресацію.

За фізичною адресою [0-0-1] (перший сектор) на диску розміщується *головний завантажувальний запис* (Master Boot Record, MBR), що містить *програму початкового завантаження* IPL1 (Initial Program Loading 1), а також *таблицю розділів* (partition table, PT).

При запуску комп’ютера, після завершення початкового тесту POST (Power-on self-test), базова система введення-виведення (BIOS) звертається до MBR. Для деяких ОС MBR містить програму stage1, тобто перший етап завантаження ОС, який завантажує програму другого етапу завантаження ОС – stage2 (іноді stage2 завантажує сектор активного розділу boot.manager або програму авторизації).

Сам по собі MBR є програмою, яку BIOS читає з жорсткого диска поміщає в ОП за фізичною адресою 0x7C00:0000, а потім передає їй керування. Завантажувальний запис продовжує процес завантаження ОС. Формат MBR наведено в табл. 1.

У MBR знаходяться три важливих елементи:

• Програма початкового завантаження. Саме вона запускається BIOS’ом після успішного завантаження в пам’ять першого сектора з MBR. Вона має розмір 446 байт і завантажує, трохи більш складну програму, звичайно стартовий сектор операційної системи і передає йому керування.

• Таблиця розділів диска (partition table). Розташовується в MBR із зсувом 1BEh і займає 64 байти.

• Сигнатура MBR. Останні два байти MBR повинні містити сигнатуру 55AAh. За наявності цієї сигнатури BIOS перевіряє, чи перший блок був завантажений успішно.

Таблиця 1 – **Формат MBR**

Зсув	Розмір	Вміст	
00h	1BEh (446)	Програма аналізу Partition Table і завантаження IPL 1 з активного розділу жорсткого диска	
1BEh	10h (16)	Розділ 1	Таблиця розділів
1CEh	10h (16)	Розділ 2	
1DEh	10h (16)	Розділ 3	
1EEh	10h (16)	Розділ 4	
1FEh	2	Ознака таблиці розділів (сигнатура) – 55AAh	

У кінці першого сектора розміщується таблиця розділів диска. Ця таблиця містить чотири елементи, що описують розміщення і характеристики розділів диска. Вона є найбільш важливою структурою даних, при її ушкодженні не буде завантажуватися ОС і стануть недоступними всі розділи і їх дані.

Можна створити резервну копію MBR в ОС Linux:

```
dd if=/dev/sda of=mbr.bin bs=512 count=1
```

Відновлення завантажувача і таблиці розділів:

```
dd if=mbr.bin of=/dev/sda bs=512 count=1
```

Відновлення тільки завантажувача:

```
dd if=mbr.bin of=/dev/sda bs=446 count=1
```

В останніх двох байтах сектора міститься число 55AAh. Це ознака таблиці розділів, формат елемента якої показано в табл. 2.

Таблиця 2 – **Формат елемента таблиці розділів MBR**

Зсув	Розмір	Вміст
Признак завантаження		
00h	1	80р – активний (завантажувальний) 00h – неактивний
Початок розділу диска (CHS)		
01h 02h 03h	1 1 1	біти 0-7 номер головки (0-255) біти 0-5 номер сектора (1-63) біти 6-7 старші біти номера циліндра біти 0-7 молодші біти номера циліндра (0-1023)
Тип розділу		
04h	1	код типу розділа
Кінець розділу диска (CHS)		
05h 06h 07h	1 1 1	Біти 0-7 номер головки (0-255) Біти 0-5 номер сектора (1-63) біти 6-7 старші біти номера циліндра біти 0-7 молодші біти номера циліндра (0-1023)
Відносний сектор (початок LBA)		
08h	4	кількість секторів перед початком розділу
Розділ		
0Ch	4	кількість секторів розділу

Типи розділів для ОС Windows і Linux показані в табл. 3.

Таблиця 3 – **Типи розділів**

Код типу розділу	Тип розділу	Код типу розділу	Тип розділу
00h	Empty (порожній розділ)	18h	AST Windows swap
01h	FAT-12	1Bh	Прихований FAT-32
04h	FAT-16 (<32 Мбайт)	1Ch	Прихований FAT-32X
05h	Розширений розділ	1Eh	Прихований FAT-16X
06h	FAT-16т	1Fh	Прихований розширений розділ LBA
07h	Windows NT NTFS	27h	Прихований NTFS (розділ відновлення системи)
0Bh	FAT-32т	82h	Linux swap
0Ch	FAT-32X, LBA	83h	Linux
0Eh	FAT-16X (VFAT), LBA	84h	Linux extended (розширений)
0Fh	Розширений розділ LBA	C2h	Прихований Linux
16h	Прихований FAT16	C3h	Прихований Linux swap
17h	Прихований NTFS		

Фізичний диск може бути розбитий на декілька логічних дисків. MBR і декілька Extended Partions Pointers (EPP) використовуються для зберігання інформації про кількість розділів на диску і їх розміщення. MBR може мати вказівник на основний розділ або на EPP блок (один сектор розміром 512 байт). EPP може мати вказівник на логічний розділ або на наступний EPP блок в ланцюжку. Використання логічних розділів дозволяє розбити диск більш яе на чотири розділи.

Якщо використовується розширений розділ, то координати початку розділу вказують на EBR (Extended Boot Record) – сектор, який описує розмір одного логічного розділу. Формат EBR показано в табл. 4.

Таблиця 4 – **Структура EBR**

Зміщення	Довжина	Описання
1BEh	16	Вказівник на розділ
1CEh	16	Вказівник на наступний EBR
1DEh	32	Не використовується
1FEh	2	Сигнатура (55AAh)

Завантажувальний запис BOOT. Перший сектор логічного диска займає завантажувальний запис (Boot Record). Цей запис зчитується з активного розділу диска програмою головного завантажувального запису MBR і запускається на виконання. Завдання завантажувального запису BOOT – виконати завантаження ОС. Кожен тип ОС має свій завантажувальний запис. Навіть для різних версій однієї і тієї ж ОС програма завантаження може виконувати різні дії. Окрім програми початкового завантаження ОС, завантажувальний запис містить параметри, що описують характеристики логічного диска. Формат запису BOOT показано в табл. 5.

Таблиця 5 – **Формат завантажувального запису BOOT**

Зсув	Розмір	Вміст
00h	3	Команда JMP xx (xx=0xE3C90 або 0xEB5890) – перехід на програму початкового завантаження IPL2
03h	8	Назва і версія ОС

0bh	2	Кількість байт в секторі (512 | 1024 | 2048 | 4096)) за замовчуванням в little endian 0x0002=> 0200=>512 байт)
0Dh	1	Кількість секторів в кластері (16)
0Eh	2	Число секторів у зарезервованій області (0x0200)
10h	1	Число FAT-32 таблиць (2)
11h	2	Максимальне число елементів у кореновому каталозі
13h	2	Число секторів у логічному розділі (65536)
15h	1	Тип переміщуваності медіа носія (0xF0- переміщуваний, 0xF8 – непереміщуваний)
16h	2	Число секторів у FAT
18h	1	Кількість секторів на доріжку (63)
1Ah	2	Значення головок для переривання 13h (255)
1Ch	4	Кількість прихованих секторів перед початком розділу (63)
20h	4	Як в 13h
24h	4	Число секторів, які використовуються в одній FAT для розділу FAT32 (9992)
28h	2	Признак дублювання входів в FAT (0-дублюються)
2Ah	2	Min і max число для FAT
2Ch	4	Номер кластеру де починається root каталог (2)
30h	2	Сектор, де знаходиться FSINFO (1), копія (7)
32h	1	Сектор, де знаходиться boot сектор (6)
34h	12	Не використовується
40h	1	Номер НЖД для перереривання 14h (0x80)
41h	1	Не використовується
42h	1	Розширена boot сигнатура (0x29 – є Serial Number, Volume Label, File System)
43h	4	Volume Serial Number
47h	11	Volume Label
52h	8	Файлова система під час форматування
5Ah	420	Програма IPL2 (boot strap)
1FEh	2	Сигнатура 0x55AA

Декілька варіантів розбиття фізичного диска на первинні і логічні розділи показано на рис. 3.

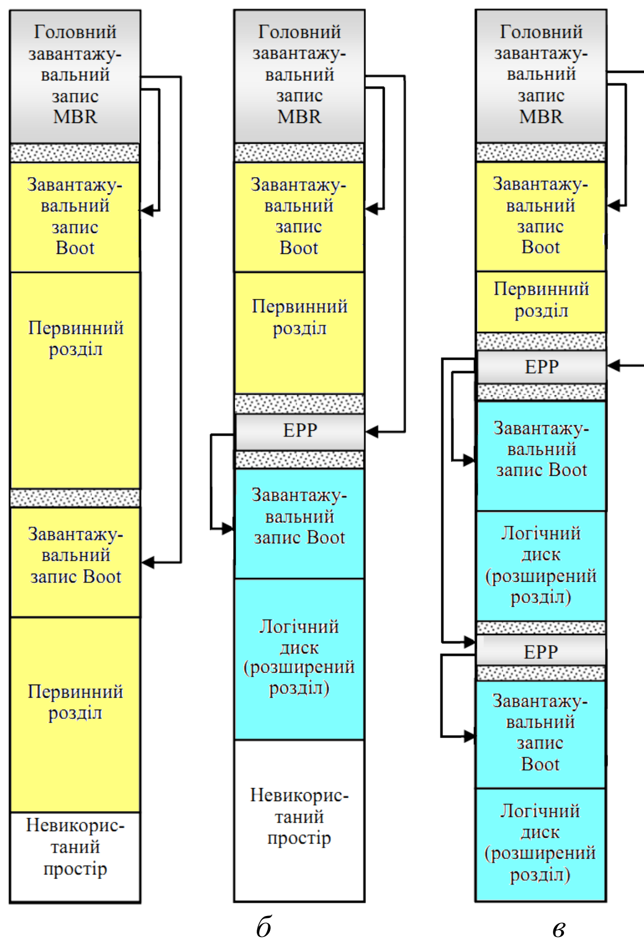

Рисунок 3 – Розділи фізичного диска:
а – первинні розділи; б – один первинний розділ і один логічний диск у розширеному розділі; в – один первинний розділ і два логічні диски (записи розширених розділів об'єднані в список)

Для задання повного шляху і імені завантажувача різних ОС використовуються спеціальні системні драйвери, як показано на рис. 4.

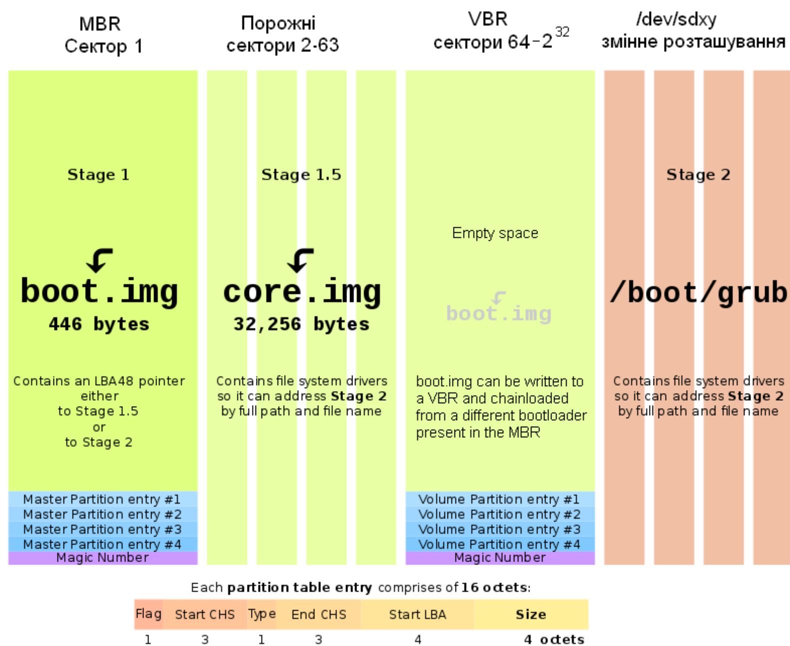

Рисунок 4 – Спеціалізований системний драйвер Grub

Таблиця розміщення файлів FAT. Існують різні таблиці розміщення файлів (File Allocation Table – FAT) FAT16, FAT32, VFAT. FAT містить інформацію, яка зв’язує кластери дискового простору з файлами.

При виконанні операцій читання-запису обмін між диском і пам’яттю здійснюється блоками. Мінімальний розмір блоку дорівнює одному сектору. Для зменшення кількості звернень до диска послідовні блоки об’єднуються в кластери. Кількість секторів в кластері дорівнює ступені двійки. Для зберігання файла відводиться ціле число кластерів (мінімум один). Кластери у які записаний файл можуть зберігатися у різних місцях диска.

Простір тому FAT32 логічно розділено на три суміжні області:

- зарезервована область, яка містить службові структури, які належать VBR;
- область таблиці FAT, яка містить масив індексних вказівників, які відповідають кластерам області даних. Для збільшення надійності на диску розміщуються дві таблиці FAT.
- область даних, де власне записано вміст файлів, а також метаінформація стосовно імен файлів і каталогів, їх атрибутів, часу створення і зміни, розмірів і розміщення на диску.

Сама таблиця FAT визначає список кластерів, в яких розміщуються файли і каталоги. Між кластерами і індексними вказівниками таблиці є взаємна однозначна відповідність. Значення індексного вказівника відповідає значенню кластера. Можливі наступні стани:

- кластер вільний (вказівник обнулений);
- кластер зайнятий файлом і не є останнім кластером файла (значення вказівника – це номер наступного кластера);
- кластер є останнім кластером файла (вказівник містить значення 0xFFFF_FFFF);
- кластер пошкоджений (вказівник містить значення 0xFFFF_FFF7);
- кластер зарезервований.

Файлові записи. Після останньої таблиці FAT розміщується область даних, яка містить файли і каталоги. Каталог є звичайним файлом, позначеним спеціальним атрибутом. Дані такого файла є списками файлових записів. Каталог не може містити файли з однаковими іменами.

Кореневий каталог. Єдиним обов’язково присутнім каталогом є кореневий каталог. Кореневий каталог є списком кластерів і має наступні особливості:

- не має позначок дати і часу;
- не має власного імені (крім “\”);
- не містить файлів з іменами “.”,”..”.
- містить файл позначки тому.

Структура файлового запису. Файловий запис FAT32 має наступну структуру:

- DIR_Name. Поле з ім’ям файла.
- DIR_Attr. Байт з атрибутами файла.
- DIR_NTRes. Байт використовується у Windows NT.
- DIR_CttlTime, DIR_CtrlDate. Час і дата створення файла.
- DIR_LstAccDate. Час останнього доступу до файла.
- DIR_FstClusHI. Номер першого кластера файла (старше слово).
- DIR_WrtTime, DIR_WrtDate. Час і дата останнього запису.
- DIR_FstClusLO. Номер першого кластера файла (молодше).
- DIR_FileSize. Розмір файла в байтах.

2.3 Стандарт GPT

В структурі MBR розділи можна адресувати в нотації C-H-S. Нотація C-H-S на даний час практично не використовується, оскільки оперує 24-бітними числами, що дозволяє описувати диски обсягом до 7,8 Гбайт. Схема LBA оперує із 32-бітними значеннями і дозволяє описувати диски до 2 Тбайт (звичайно розмір сектора дорівнює 512 байт). При використанні MBR виникають проблеми цілістності даних, так як логічні розділи мають

структуру зв'язаного списку. Пошкодження одного із розділів може заблокувати доступ до інших логічних розділів.

Усунути вказані обмеження і недоліки дозволяє *стандарт формату розміщення таблиці розділів GPT* (GUID Partition Table). Він є частиною *стандарту EFI* (Extensible Firmware Interface), запропонованого фірмою Intel на заміну BIOS. EFI використовує GPT там, де BIOS використовує MBR.

GPT використовує сучасну систему адресації логічних блоків (LBA) замість застосовуваної в MBR адресації C-H-S. GPT містить MBR на самому початку диска (блок LBA 0) як для захисту, так і з метою сумісності, а заголовок GPT міститься в блоці LBA 1. Заголовок містить адресу блоку де починається сама таблиця розділів, звичайно це наступний блок LBA 2. Для 64-розрядної версії Windows за таблицею розділів зарезервовано 16384 байт (для секторів розміром 512 байтів це буде 32 сектори), так що першим використовуваним сектором кожного жорсткого диска буде блок LBA 34.

Крім того GPT забезпечує дублювання – заголовок і таблиця розділів записані як на початку, так і вкінці диска. Теоретично GPT дозволяє створювати розділи диска розміром $9{,}4\times10^{21}$ байт, в той час як MBR може працювати тільки до $2{,}2\times10^{12}$ байт. На рис. 5 показана схема, яка пояснює формат GPT.

LBA 0	Захищений MBR
LBA 1	Заголовок основного GPT
LBA 2	Вхід 1 Вхід 2 Вхід 3 Вхід 4
LBA 3	Входи 5-128
LBA34	Розділ 1
	Розділ 2
	...
LBA - 34	Розділ N
LBA - 33	Вхід 1 Вхід 2 Вхід 3 Вхід 4
	Входи 5-128
LBA - 2	
LBA - 1	Заголовок, вторинного GPT

Рисунок 5 – Схема формату GPT: розмір блоку LBA 512 байт, від'ємна адресація блоків вказує, що їх нумерація починається з кінця

2.4 Неперервне розміщення файлів

Найпростіший підхід до фізичної організації файлових систем – це неперервне розміщення файлів. При цьому кожному файлові відповідає набір неперервно розміщених кластерів на диску (рис. 6). Для кожного файла має зберігатися адреса початкового кластера і розмір файла.

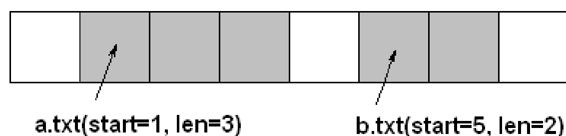

Рисунок 6 – Неперервне розміщення файлів

Зазначимо, що розподіл дискового простору в цьому разі подібний до динамічного розподілу пам'яті. Для пошуку вільного блоку на диску можна використати алгоритми першого придатного або найкращого придатного блоку.

Неперервне розміщення файлів вирізняється простотою в реалізації та ефективністю (наприклад, весь файл може бути зчитаний за одну операцію), але має істотні недоліки.

• Під час створення файла користувач має заздалегідь задати його максимальну довжину і виділити весь простір на диску за один раз. Збільшувати розміри файлів під час роботи не можна. У багатьох ситуаціях це абсолютно неприйнятно (наприклад, неможливо вимагати від користувача текстового редактора щоб він вказував остаточну довжину файла перед його редагуванням).

• Вилучення файлів згодом може спричинити велику зовнішню фрагментацію дискового простору з тих самих причин, що й за динамічного розподілу пам'яті.

У сучасних ОС для організації даних на жорстких дисках неперервне розміщення майже не використовують, проте його застосовують у таких файлових системах, де можна заздалегідь передбачити, якого розміру буде файл. Прикладом є файлові системи для компакт-дисків. Вони мають кілька властивостей, що роблять неперервне розміщення файлів найкращим рішенням:

• така файлова системи записується повністю за один раз, так як під час запису для кожного файла наперед відомий його розмір;

• файлові системи на компакт-диска лише читаються, файли в них ніколи не розширюються і не вилучаються, тому відсутні причини появлення зовнішньої фрагментації.

2.5 Розміщення файлів зв'язаними списками

Прості зв'язані списки. Іншим підходом є організація кластерів файла у зв'язаний список. Кожен кластер файла містить інформацію про те, де перебуває наступний кластер цього файла (наприклад, його номер). Найпростіший приклад такого розміщення показано на рис. 7. Заголовок файла в цьому випадку має містити посилання на його перший кластер, вільні кластери можуть бути організовані в аналогічний список.

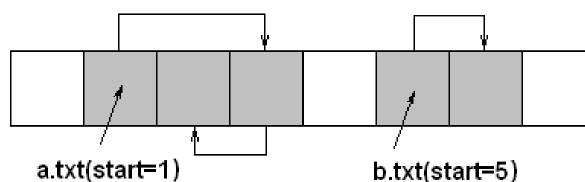

Рисунок 7 – Найпростіший приклад зв'язаного розміщення файлів

Розміщення файлів з використанням зв'язаних списків надає такі переваги:

• відсутність зовнішньої фрагментації (є тільки невелика внутрішня фрагментація, пов'язана з тим, що розмір файла може не ділитися націло на розмір кластера);

• мінімум інформації, яка потрібна для зберігання у заголовку файла (тільки посилання на перший кластер);

• можливість динамічної зміни розміру файла;

• простота реалізації керування вільними блоками, яка принципово не відрізняється від керування розміщенням файлів.

Цей підхід, однак, не позбавлений і серйозних недоліків:

• відсутність ефективної реалізації випадкового доступу до файла: для того щоб одержати доступ до кластера з номером *n*, потрібно прочитати всі кластери файла з номерами від 1 до *n*-1;

• зниження продуктивності тих застосувань, які зчитують дані блоками, за розміром рівними ступеню числа 2 (а таких застосувань досить багато): частина будь-якого кластера

повинна містити номер наступного, тому корисна інформація в кластері займає обсяг, не кратний його розміру (цей обсяг навіть не є ступенем числа 2);

• можливість втрати інформації у послідовності кластерів: якщо внаслідок збою буде втрачено кластер на початку файла, вся інформація в кластерах, що йдуть за ним, також буде втрачена.

Є модифікації цієї схеми, які зберегли своє значення дотепер, найважливішою з них є використання таблиці розміщення файлів.

Зв'язані списки з таблицею розміщення файлів. Цей підхід полягає в тому, що всі посилання, які формують списки кластерів файла, зберігаються в окремій ділянці файлової системи фіксованого розміру, формуючи *таблицю розміщення файлів* (File Allocation Table, FAT), рис. 8. Елемент такої таблиці відповідає кластеру на диску і може містити:

• номер наступного кластера, якщо цей кластер належить файлу і не є його останнім кластером;

• індикатор кінця файла, якщо цей кластер є останнім кластером файла;

• індикатор, який показує, що цей кластер вільний.

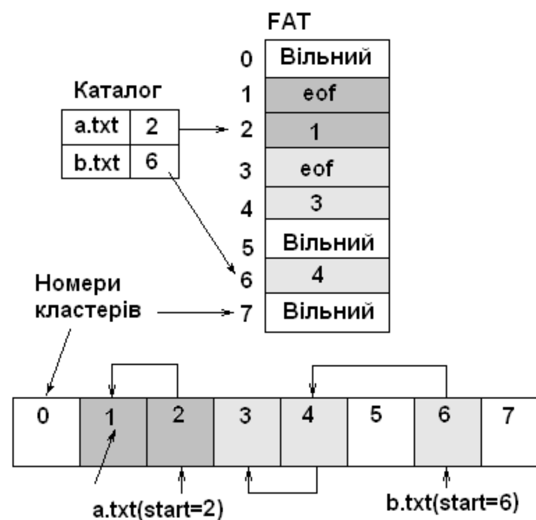

Рисунок 8 – Використання таблиці розміщення кластерів

Для організації файла достатньо помістити у відповідний йому елемент каталогу номер першого кластера файла. За необхідності прочитати файл система знаходить за цим номером кластера відповідний елемент FAT, зчитує із нього інформацію про наступний кластер і т. д. Цей процес триває доти, поки не зустрінеться індикатор кінця файла.

Використання цього підходу дає змогу підвищити ефективність і надійність розміщення файлів зв'язними списками. Це досягається завдяки тому, що розміри FAT дозволяють кешувати її в пам'яті. Через це доступ до диска під час відстеження посилань заміняють звертаннями до оперативної пам'яті. Зазначимо, що навіть якщо таке кешування не реалізоване, випадковий доступ до файла не призводитиме до читання всіх попередніх його кластерів – зчитані будуть тільки попередні елементи FAT.

Крім того, спрощується захист від збоїв. Для цього, наприклад, можна зберігати на диску додаткову копію FAT, що автоматично синхронізуватиметься з основною. У разі ушкодження однієї з копій інформація може бути відновлена з іншої. І нарешті, службову інформацію більше не зберігають безпосередньо у кластерах файла, вивільняючи в них місце для даних. Тепер обсяг корисних даних всередині кластера майже завжди (за винятком, можливо, останнього кластера файла) дорівнюватиме ступеню числа 2.

Однак, у разі такого способу розміщення файлів для розділів великого розміру обсяг FAT може стати доволі великим і її кешування може потребувати значних витрат пам'яті. Скоротити розмір таблиці можна, збільшивши розмір кластера, але це, в свою чергу,

призводить до збільшення внутрішньої фрагментації для малих файлів (менших за розмір кластера).

Також руйнування обох копій FAT (внаслідок апаратного збою або дії програми-зловмисника, наприклад, комп'ютерного вірусу) робить відновлення даних дуже складною задачею, яку не завжди можна розв'язати.

2.6 Індексоване розміщення файлів

Базовою ідеєю ще одного підходу до розміщення файлів є перелік адрес всіх кластерів файла в його заголовку. Такий заголовок файла дістав назву *індексного дескриптора*, або і-вузла (`inode`), а сам підхід – індексованого розміщення файлів.

За індексованого розміщення із кожним файлом пов'язують його `inode`. Він містить масив із адресами (або номерами) усіх кластерів цього файла, при цьому *n*-й елемент масиву відповідає *n*-му кластеру. Індексні дескриптори `inode` зберігають окремо від даних файла, для цього звичайно виділяють на початку розділу спеціальну ділянку індексних дескрипторів `inode`. В елементі каталогу розміщують номер індексного дескриптора відповідного файла (рис. 9).

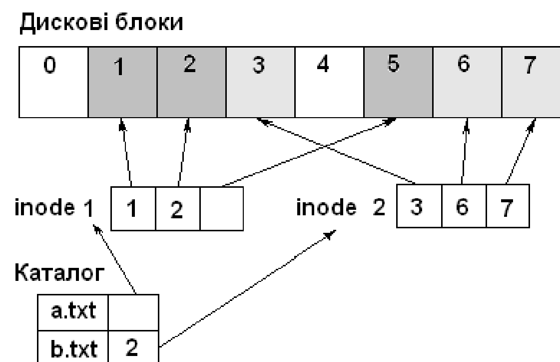

Рисунок 9 – Індексоване розміщення файлів

Під час створення файла на диску розміщують його індексний дескриптор `inode`, у якому всі вказівники на кластери спочатку є порожніми. Під час першого записування в *n*-й кластер файла менеджер вільного простору виділяє вільний кластер і його номер або адресу заносять у відповідний елемент масиву.

Цей підхід стійкий до зовнішньої фрагментації й ефективно підтримує як послідовний, так і випадковий доступ (інформація про всі кластери зберігається компактно і може бути зчитана за одну операцію). Для підвищення ефективності індексний дескриптор повністю завантажують у пам'ять, коли процес починає працювати з файлом, і залишають у пам'яті доти, поки ця робота триває.

Структура індексних дескрипторів `inode`. Основною проблемою є підбір розміру і встановлення оптимальної структури індексного дескриптора `inode`, оскільки:

• з одного боку, зменшення розміру дескриптора може значно зекономити дисковий простір і пам'ять (дескрипторів потрібно створювати значну кількість – по одному на кожний файл, разом вони можуть займати досить багато місця на диску; крім того, для кожного відкритого файла дескриптор буде розташовано в оперативній пам'яті).

• з іншого боку, дескриптора надто малого розміру може не вистачити для розміщення інформації про всі кластери великого файла.

Одне з компромісних розв'язань цієї задачі, яке застосовується вже багато років у UNIX-системах, показане на рис. 10. Під час його опису замість терміна "кластер" вживається його синонім "дисковий блок".

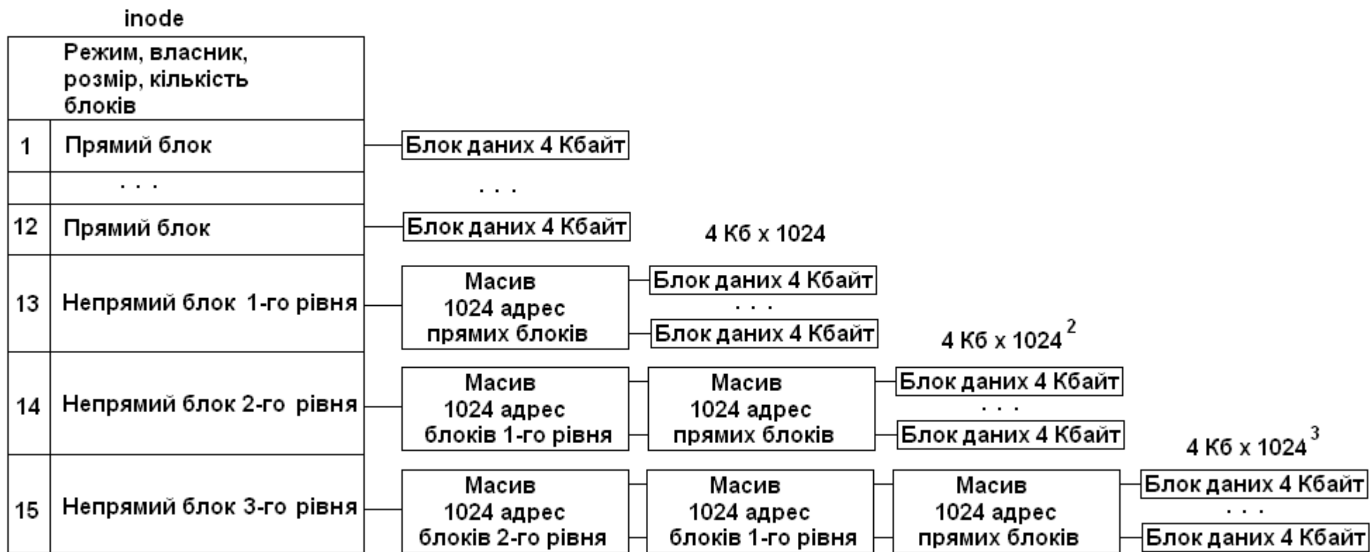

Рисунок 10 – Структура індексного дескриптора `inode`

У цьому випадку індексний дескриптор містить елементи різного призначення:

• Частина елементів (звичайно перші 12) безпосередньо вказує на дискові блоки, які називають прямими (direct blocks). Отже, якщо файл може вміститися у 12 дискових блоках (за розміру блоку 4 Кбайт максимальний розмір такого файла становитиме 4096 × 12 = 49_152 байти), усі ці блоки будуть прямо адресовані його індексним дескриптором `inode` і жодних додаткових структур даних не буде потрібно.

• Якщо файлу необхідно для розміщення даних більше, ніж 12 дискових блоків, використовують непряму адресацію першого рівня. У цьому випадку 13-й елемент індексного дескриптора вказує не на блок із даними, а на спеціальний непрямий блок першого рівня (single indirect block). Він містить масив адрес наступних блоків файла (за розміру блоку 4 Кбайт, а адреси – 4 байти в ньому міститимуться адреси 1024 блоків, при цьому максимальний розмір файла буде 4096 × (12+1024) = 4_234_456 байт).

• Якщо файлу потрібно для розміщення більше ніж 1024 + 12 = 1036 дискових блоків, використовують непряму адресацію другого рівня. 14-й елемент індексного дескриптора `inode` в цьому разі вказуватиме на непрямий блок другого рівня (double indirect block). Такий блок містить масив з 1024 адрес непрямих блоків першого рівня, кожен із них, як зазначалося, містить масив адрес дискових блоків файла. Тому за допомогою такого блоку можна адресувати 1024^2 додаткових блоків.

• Нарешті, якщо файлу потрібно більше ніж $1036 + 1024^2$ дискових блоків, використовують непряму адресацію третього рівня. Останній 15-й елемент індексного дескриптора `inode` вказуватиме на непрямий блок третього рівня (triple indirect block), що містить масив з 1024 адрес непрямих блоків другого рівня, даючи змогу адресувати додатково 1024^3 дискових блоків.

Розріджені файли. Багато операційних систем не зберігають вказівники на дискові блоки файлів у їхніх індексних дескрипторах, поки до них не було доступу для записування. Фрагменти, до яких цього доступу не було з моменту створення файла, називають “дірками” (holes), дисковий простір під них не виділяють, але під час розрахунку довжини файла їх враховують. У разі читання вмісту “дірки” повертають блоки, заповнені нулями, звертання до диска не відбувається.

На практиці “дірки” найчастіше виникають, коли вказівник поточної позиції файла переміщують далеко за його кінець, після чого виконують операцію записування. У результаті розмір файла збільшується без додаткового виділення дискового простору. Подібні файли називають розрідженими файлами (sparse files). Вони реально займають на

диску місця набагато менше, ніж їхня довжина, фактично довжина розрідженого файла може перевищувати розмір розділу, на якому він перебуває.

3 Файлова система NTFS

Файлова система NTFS. Особливості ФС NTFS:

– спроектована спеціально для Windows;

– підтримує транзакції;

– всі дані зберігаються у файлах;

– підтримує 64-бітові вказівники для структур даних;

– підтримує імена файлів до 255 символів (повний шлях до файла до 32767 символів) і кодувань Unicode;

– підтримує стиснення;

– підтримує шифрування (EFS);

– стійка до відмов;

– підтримує декілька потоків даних для одного файла.

Логічні та віртуальні номери кластерів NTFS. Файлова система NTFS працює з цілим числом дискових секторів як з мінімальним одиничним блоком даних. Такий блок називають *кластером*. Розмір кластера визначаюється під час форматування, тому різні томи можуть мати різні розміри кластерів. Файлова система обчислює розмір кластера, враховуючи розмір диска та тип використовуваної ФС. Кластер може мати розмір 1-64 Кбайт.

До кластерів належать декілька важливих параметрів NTFS. Перший параметр називають *логічним номером кластера* (Logical Cluster Number – LCN). Файлова система NTFS ділить диск на кластери й призначає кожному кластеру номер, починаючи з нуля. Цей номер називають LCN.

Іншим важливим параметром є *віртуальний номер кластера* (Virtual Cluster Number – VCN), який вказує номер кластера всередині певного файла. Віртуальний номер кластера дозволяє обчислити місцезнаходження атрибута, наприклад зсув даних усередині файла, а логічний номер кластера дає змогу обчислити зсув відповідно тому або розділу для певного блока даних.

Максимальний розмір розділу NTFS обмежений лише розмірами жорстких дисків.

Диск NTFS умовно поділяють на дві частини. Перші 12% диска відводяться під зону MFT – ділянку, в якій розміщується нарощення метафайла MFT. Запис будь-яких даних в цю ділянку неможливий. Зона MFT завжди утримується порожньою – це робиться для того, щоб найголовніший, службовий файл (MFT) не фрагментувався у разі свого зростання. Інші 88 % диска є звичайним простором для зберігання файлів.

Фізична структура NTFS. Фізичну структуру NTFS показано на рис. 11.

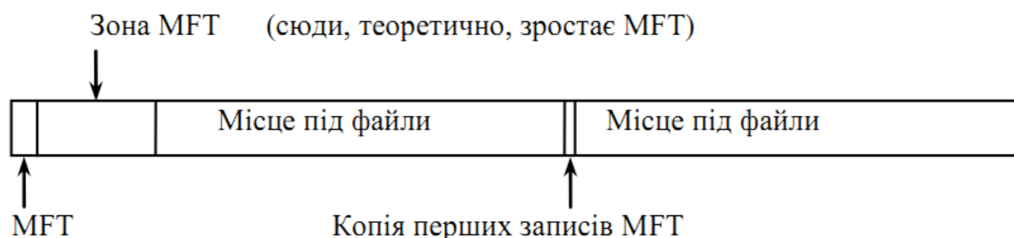

Рисунок 11 – Фізична структура NTFS

Вільне місце на диску включає все фізично вільне місце й незаповнені зони MFT. Механізм використання зони MFT такий: коли файли вже не можна записувати в звичайну ділянку, зона MFT просто скорочується, звільняючи таким чином місце для запису файлів. У разі звільнення місця в звичайній ділянці зона MFT може знову розширитись. При цьому не виключена ситуація, що в цій зоні залишилися й звичайні файли.

MFT та її структура. Файлова система NTFS є значним досягненням структуризації: кожен елемент системи є файл – навіть службова інформація.

Найголовніший файл NTFS називається MFT (Master File Table) – головна таблиця файлів. Саме він розміщується в зоні MFT і є централізованим каталогом решти всіх файлів диска та себе самого.

Файл MFT поділено на записи фіксованої місткості (звичайно 1 Кбайт), і кожен запис відповідає якому-небудь файлу. Перші 16 файлів мають службове призначення і недоступні ОС – вони називаються метафайлами (табл. 6), причому найперший метафайл – сама MFT. Ці перші 16 елементів MFT – єдина частина диска, що має фіксоване положення. Решта MFT файла може розміщуватися, як і будь-який інший файл, у довільних місцях диска – відновити його положення можна за допомогою його самого, «зачепившись» за саму основу – за перший елемент MFT.

Таблиця 6 – Метафайли

Файл	Номер запису	Опис
$Mft	0	Головна файлова таблиця
$MftMirr	1	Дзеркало MFT, що містить копію перших 16 файлів MFT
$LogFile	2	Файл журналу (для відновлення після збоїв і підтримання цілісності ФС)
$Volume	3	Опис тому, включаючи серійний номер тому, дату та час створення, а також прапор тому
$AttrDef	4	Визначення атрибута
. (крапка)	5	Кореневий каталог
$Bitmap	6	Бітова карта розміщення кластерів
$Boot	7	Завантажувальний запис диска
$BadClus	8	Список пошкоджених кластерів
$Quota $Secure	9	У Windows NT 4 визначений як файл призначених для користувача квот, проте ніколи не використовувався. У Windows 2000 перевизначений як дескриптор безпеки
$UpCase	10	Таблиця верхнього реєстра
$Extend	11	Каталог, який містить файли $Objid, $Quota і $UsrJrnl. Використовується в Windows 2000 і пізніших версіях
--	12-23	Зарезервовані

Файли і потоки. Файлова система NTFS – це файл-об’єкт, що містить файли-об’єкти.

Файл-об’єкт щонайменше має запис в MFT. У цьому місці зберігається вся інформація про файл, за винятком власних даних. Ім’я файла, розмір, положення на диску окремих фрагментів і т. ін. Якщо для інформації не вистачає одного запису MFT, то використовуються декілька, причому не обов’язково підряд.

У NTFS підтримується декілька потоків даних для одного файла. Потік можна відкрити за допомогою функції Win32 API `CreateFile`, а ім’я потоку може бути додано до імені файла, наприклад `File1:Stream25`. Потоки підтримують запис, зчитування та незалежне від інших відкритих потоків блокування.

Атрибути файлів NTFS. Як атрибут можна вказати ім’я файла, список керування доступом до файла й дані файла (рис. 12). Якщо дані атрибуту мають невеликий розмір, вони будуть збережені безпосередньо в записі MFT, такі атрибути називають резидентними.

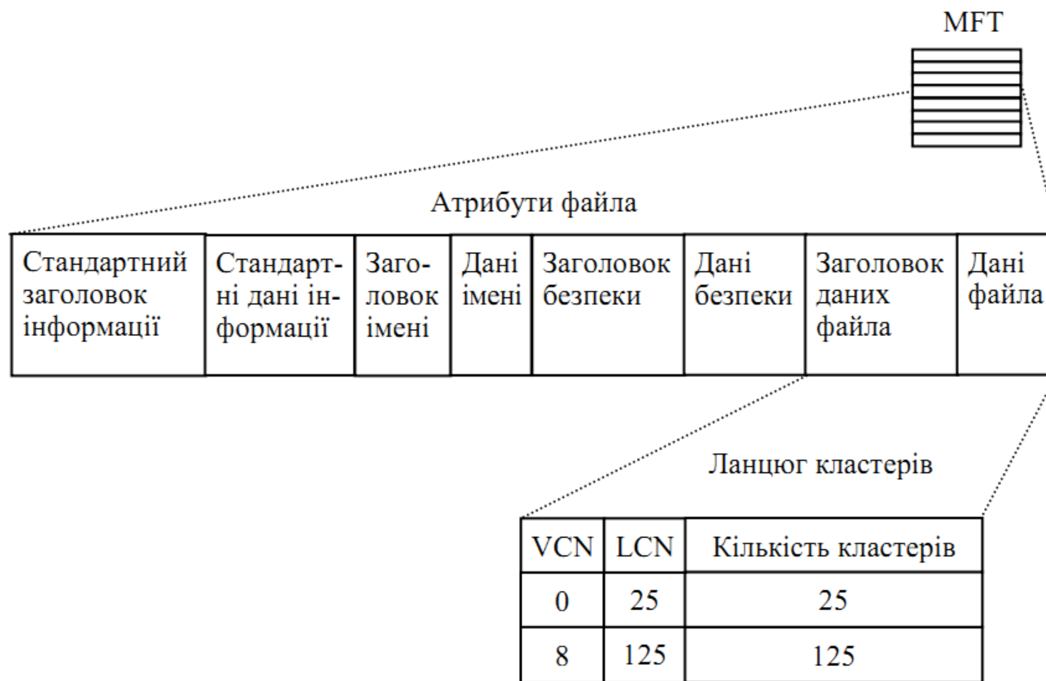

Рисунок 12 – Атрибути файлів

Якщо дані завеликі для зберігання в MFT, зберігається інформація про розміщення цих даних (номери кластерів, у яких розміщені необхідні дані). Такі атрибути називають нерезидентними.

Деякі атрибути файлів та їх опис наведено в табл. 6.

Таблиця 6 – Атрибути файлів

Тип атрибута	Опис
Standard Information (стандартна інформація)	Включає бюджет зв'язку
Attribute List (список атрибутів)	Перераховує всі інші атрибути (тільки у великих файлах)
Filename (ім'я файла)	Атрибут, що повторюється для довгих і коротких імен файлів. Довге ім'я файла може містити до 255 символів Unicode. Коротке ім'я, — доступне для MS-DOS, вісім плюс три символи, без урахування регістра. Додаткові імена, або жорсткі зв'язки (hard links), використовуються POSIX і можуть бути також включені як додаткові атрибути імені файла
Security Descriptor (дескриптор безпеки)	Фіксує інформацію про того, хто може звертатися до файла, хто є його власником і т. ін.
Data (дані)	Містить дані файла
Index Root (корінь індексів)	Використовується під час роботи з каталогами
Index Root (корінь індексів)	Використовується під час роботи з каталогами
Index Allocation (індексне розміщення)	Використовується під час роботи з каталогами
Volume Information (інформація тому)	Використовується тільки в системному файлі тому і включає зокрема версію та ім'я тому
Bitmap	Надає інформацію про використання записів в MFT або

(бітовий масив)	каталозі
Extended Attribute Information (інформація розширеного атрибуту)	Використовується файловими серверами, які пов'язані із системами OS/2. Цей тип атрибута не використовується Windows NT
Extended Attributes (розширені атрибути)	Використовується файловими серверами, які пов'язані із системами OS/2. Цей тип атрибута не використовується Windows NT

Каталоги. Каталог у NTFS є специфічним файлом, що зберігає посилання на інші файли та каталоги, створюючи ієрархічну структуру даних на диску.

Файл каталога поділений на блоки, кожний з яких містить ім'я файла, базові атрибути та посилання на елемент MFT, який вже надає повну інформацію про елемент каталога. Внутрішня структура каталога є бінарним деревом.

Ведення журналу. Відмовостійка система NTFS цілком може набути коректного стану у разі будь-яких реальних збоїв. Будь-яка сучасна ФС ґрунтується на такому понятті, як транзакція – дія, що виконується цілком і коректно або не виконується взагалі. У NTFS просто не буває проміжних (помилкових або некоректних) станів – квант зміни даних не може бути поділений на до і після збоїв – він або здійснений, або відмінений.

4 Файлові системи UNIX

Файлові системи UNIX. У ФС UNIX System V Release 4 реалізований механізм віртуальної ФС VFS (Virtual File System), який дозволяє ядру системи одночасно підтримувати декілька різних типів ФС.

Типи файлових систем, підтримуваних в UNIX System V Release 4:

s5 – традиційна ФС UNIX System V;

ufs – ФС, використовувана за замовчуванням в UNIX System V Release 4;

nfs – адаптація відомої ФС NFS фірми Sun Microsystems, яка дозволяє розділяти файли та каталоги в гетерогенних мережах;

rfs – ФС Remote File Sharing з UNIX System V Release 3. За функціональними можливостями близька до NFS, але вимагає на кожному комп'ютері встановлення UNIX System V Release 3 або пізніших версій цієї ОС;

veritas – відмовостійка ФС з транзакційним механізмом операцій;

specfs – новий тип ФС, що забезпечує єдиний інтерфейс до всіх спеціальних файлів, що описуються в каталозі /dev;

fifofs – нова ФС, що використовує механізм VFS для реалізації файлів FIFO, відомих також як конвеєри (pipes), у середовищі STREAMS;

bfs – завантажувальна ФС. Призначена для швидкого і простого завантаження і тому є дуже простою плоскою ФС, що складається з одного каталога;

/proc – ФС цього типу забезпечує доступ до образу адресного простору кожного активного процесу системи, звичайно використовується для налагодження і трасування;

/dev/fd – тип ФС забезпечує зручний метод посилань на дескриптори відкритих файлів.

Традиційна файлова система S5. Типи файлів. Файлова система UNIX s5 підтримує логічну організацію файла у вигляді послідовності байтів. За функціональним призначенням розрізняються звичайні файли, каталоги та спеціальні файли. Звичайні файли містять ту інформацію, яку заносить у них користувач або яка утворюється в результаті роботи системних і призначених для користувача програм, тобто ОС не накладає ніяких обмежень на структуру та характер інформації, що зберігається в звичайних файлах.

Каталог – файл, що містить службову інформацію ФС про групу файлів, що входять у цей каталог. У каталог можуть входити звичайні, спеціальні файли і каталоги нижчого рівня.

Спеціальний файл – фіктивний файл, що асоціюється з будь-яким пристроєм введення-виведення, використовується для уніфікації механізму доступу до файлів і зовнішніх пристроїв.

Структура ФС. Файлова система s5 має ієрархічну структуру, в якій рівні створюються за рахунок каталогів, що містять інформацію про файли нижчого рівня.

Кореневий каталог ФС завжди розміщується на системному пристрої (диск, що має таку ознаку). Проте це не означає, що й решта файлів може міститися тільки на ньому. Для зв'язку ієрархій файлів, розміщених на різних носіях, застосовується монтування ФС, що виконується системним викликом `mount`.

Операція монтування наступна: у кореневій ФС вибирається деякий існуючий каталог, що містить один порожній файл. Після виконання монтування вибраний каталог стає кореневим каталогом іншої ФС. Через цей каталог змонтована ФС приєднується як гілка до загального дерева.

Привілеї доступу. В UNIX s5 всі користувачі за доступом до файла діляться на три категорії: власник, член групи власника та всі інші.

Група – це користувачі, які об'єднані за якою-небудь ознакою, наприклад, за належністю до однієї розробки. Окрім цього, в системі існує суперкористувач, що має абсолютні права доступу до всіх файлів системи.

Визначено три види доступу до файла – зчитування, запис і виконання. Привілеї доступу до кожного файла визначені для кожної з трьох категорій користувачів і для кожної з трьох операцій доступу. Початкові значення прав доступу до файла встановлюються під час його створення ОС і можуть змінюватися його власником або суперкористувачем.

Фізична організація файла. У загальному випадку файл може розміщуватися в несуміжних блоках дискової пам'яті. Логічна послідовність блоків у файлі задається набором з 13 елементів. Перші 10 елементів призначаються для безпосередньої вказівки номерів перших 10 блоків файла. Якщо розмір файла перевищує 10 блоків, то в 11-му елементі вказується номер блока, у якому міститься список наступних 128 блоків файла. Якщо файл має розмір більший, ніж 10+128 блоків, то використовується 12-й елемент, що містить номер блока, у якому вказуються номери 128 блоків, кожен з яких може містити ще по 128 номерів блоків файла. Таким чином, 12-й елемент використовується для дворівневої непрямої адресації. У випадку, якщо файл більший, ніж 10+128+1282 блоки, то використовується 13-й елемент для трирівневої непрямої адресації. За такого способу адресації граничний розмір файла становить 2_113_674 блоки. Традиційна ФС s5 підтримує розміри блоків 512, 1024 або 2048 байт.

Інформація про файл. Індексні дескриптори. Уся необхідна ОС інформація про файл, окрім його символьного імені, зберігається в спеціальній системній таблиці – індексному дескрипторі (inode) файла. Індексні дескриптори всіх файлів мають однакову ємність – 64 байт і містять дані про тип файла, фізичне розміщення файла на диску (описані вище 13 елементів), ємність у байтах, дату створення останньої модифікації та останнього звернення до файла, привілеї доступу та деяку іншу інформацію. Індексні дескриптори пронумеровані й зберігаються в спеціальній ділянці ФС. Номер індексного дескриптора є унікальним іменем файла. Відповідність між повними символьними іменами файлів і їх унікальними іменами встановлюється за допомогою ієрархії каталогів.

Структура каталога. Каталог є сукупністю записів про всі файли та каталоги, що входять до нього. Кожен запис складається з 16 байт; 14 байт відводиться під коротке символьне ім'я файла або каталога, а 2 байт – під номер індексного дескриптора цього файла. У каталозі файлової системи s5 безпосередньо не вказуються характеристики файлів. Така організація ФС дозволяє з найменшими витратами перебудовувати систему каталогів. Наприклад, у разі добавлення або вилучення файла з каталога відбувається маніпулювання меншими обсягами інформації. Крім того, для добавлення одного й того ж файла в різні каталоги не потрібно мати декількох копій як характеристик, так і самих файлів. Для цього в індексному дескрипторі ведеться облік посилань на цей файл із всіх каталогів. Як тільки кількість посилань дорівнюватиме нулю, індексний дескриптор цього файла знищується.

Структура диска s5. Весь дисковий простір, відведений під ФС, поділяють на чотири блоки (рис. 13):

– *завантажувальний блок* (boot), у якому зберігається завантажувач ОС;

– *суперблок* (superblock) – містить найзагальнішу інформацію про ФС: розмір ФС, розмір блока індексних дескрипторів, кількість індексних дескрипторів, список вільних блоків і список вільних індексних дескрипторів, а також іншу адміністративну інформацію;

– *блок індексних дескрипторів*, порядок розміщення індексних дескрипторів у якому відповідає їх номерам;

– *блок даних*, у якому розміщені як звичайні файли, так і файли-каталоги. Спеціальні файли подані у ФС тільки записами у відповідних каталогах та індексними дескрипторами спеціального формату, але місця в блоці даних не займають.

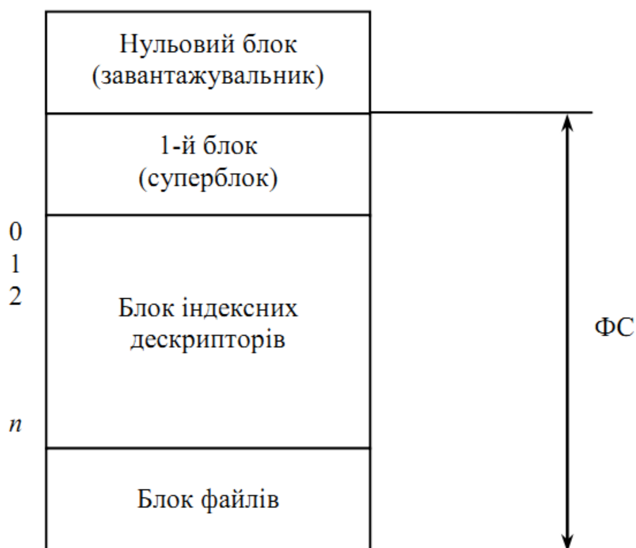

Рисунок 13 – Структура диска s5

Доступ до файла. Доступ до файла здійснюється послідовним переглядом усього ланцюга каталогів, що входять у повне ім'я файла, та відповідних їм індексних дескрипторів. Пошук завершується після отримання всіх характеристик з індексного дескриптора заданого файла. Ця процедура вимагає в загальному випадку декількох звернень до диска пропорційно до кількості складових у повному імені файла. Для зменшення середнього часу доступу до файла його дескриптор копіюється в спеціальну системну ділянку пам'яті. Копіювання індексного дескриптора входить до процедури відкриття файла.

Відкриття файла. Для відкриття файла ядро виконує такі дії:

– перевіряє існування файла; якщо файла немає, то чи можна його створити. Якщо він є, то чи дозволений до нього доступ необхідного вигляду;

– копіює індексний дескриптор з диска в оперативну пам'ять; якщо з указаним файлом уже ведеться робота, то нова копія індексного дескриптора не створюється;

– створює в ділянці ядра структуру, призначену для відображення поточного стану операції обміну даними з указаним файлом. Ця структура, названа `file`, містить дані про тип операції (зчитування, запис або зчитування й запис), про кількість зчитаних або записаних байтів, вказівник на байт файла, з яким виконується операція;

– робить позначку в контексті процесу, що видав системний виклик на операцію з файлом.

Віртуальна файлова система VFS. Ідеологія VFS. Система VFS не орієнтується на яку-небудь конкретну ФС, механізми реалізації ФС повністю приховані як від користувача, так і від застосунків. В ОС немає системних викликів, призначених для роботи зі специфічними типами ФС, а є абстрактні виклики типу `open` (відкриття), `read` (зчитування), `write` (запис) та інші, які мають змістовний опис, узагальнюючий деяким чином зміст цих операцій в найбільш

популярних типах ФС (наприклад, s5, ufs, nfs і т.п.). Система VFS також надає ядру можливість операції з ФС як з єдиним цілим: операції монтування та демонтування, а також операції отримання загальних характеристик конкретної ФС (розміру блока, кількості вільних і зайнятих блоків і т. ін.) у єдиній формі. Якщо конкретний тип ФС не підтримує якоїсь абстрактної операції VFS, то ФС повинна повернути ядру код повернення, що сповіщає про цей факт.

Інформація про файли та типи файлів VFS. У VFS вся інформація про файли розділена на дві частини — незалежну від типу ФС, яка зберігається в спеціальній структурі ядра — структурі `vnode`, і залежну від типу ФС — структура `inode`, формат якої на рівні VFS не визначений, а використовується тільки посилання на неї в структурі `vnode`. Ім'я `inode` не означає, що ця структура збігається зі структурою індексного дескриптора inode ФС s5.

Віртуальна ФС VFS підтримує такі типи файлів:

– звичайні файли;

– каталоги;

– спеціальні файли;

– іменовані конвеєри;

– символьні зв'язки.

Змістовний опис звичайних файлів, каталогів і спеціальних файлів та зв'язків не відрізняється від їх опису у ФС s5.

Символьні зв'язки. М'який зв'язок, названий символьним зв'язком і реалізується за допомогою системного виклику `symlink`.

Символьний зв'язок – це файл даних, що містить ім'я файла, з яким передбачається встановити зв'язок. Символьний зв'язок може бути створений навіть з неіснуючим файлом. Під час створення символьного зв'язку утворюється як новий вхід у каталг, так і новий індексний дескриптор `inode`. Окрім цього, резервується окремий блок даних для зберігання повного імені файла, на який він посилається.

Є три системні виклики, які стосуються символьних зв'язків:

– *readlink* – зчитування повного імені файла або каталогу, на який посилається символьний зв'язок. Ця інформація зберігається в блоці, пов'язаному із символьним зв'язком;

– *lstat* – аналогічний системному виклику `stat`, але використовується для отримання інформації про сам зв'язок;

– *lchown* – аналогічний системному виклику `chown`, але використовується для зміни власника самого символьного зв'язку.

Реалізація файлової системи VFS. UNIX System V Release 4 має масив структур vfssw, кожна з яких описує ФС конкретного типу, яка може бути встановлена в системі. Структура vfssw складається з чотирьох полів:

– символьного імені ФС;

– вказівника на функцію ініціалізації ФС;

– вказівника на структуру, що описує функції, які реалізують абстрактні операції VFS у цій конкретній ФС;

– прапорів, які не використовуються в описуваній версії UNIX.

Операції із файловою системою. Операції, виконувані з ФС VFS, наведено в табл. 6.

Таблиця 6 – Операції із ФС

VFS_MOUNT	Монтування
VFS_UNMOUNT	Розмонтування
VFS_ROOT	Отримання vnode для кореня
VFS_STATVFS	Отримання статистики
VFS_SYNC	Виштовхування буферів на диск
VFS_VGET	Отримання vnode за номером дескриптора файла
VFS_MOUNTROOT	Монтування кореневої ФС

Абстрактні операції із файлами. Абстрактні операції, виконувані із файлами, наведено в табл. 7.

Таблиця 7 – Абстрактні операції із файлами

VOP_OPEN	Відкрити файл
VOP_CLOSE	Закрити файл
VOP_READ	Зчитати з файла
VOP_WRITE	Записати в файл
VOP_IOCTL	Керувати введенням/виведенням
VOP_SETFL	Встановити прапори статусу
VOP_GETATTR	Отримати атрибути файла
VOP_SETATTR	Встановити атрибути файла
VOP_LOOKUP	Знайти vnode за іменем файла
VOP_CREATE	Створити файл
VOP_REMOVE	Вилучити файл
VOP_LINK	Зв’язати файл
VOP_MAP	Відобразити файл у пам’ять

Структура vnodeops. Окрім операцій із ФС, для кожного типу ФС (s5, ufs), установленої в ОС, необхідно описати спосіб реалізації абстрактних операцій із файлами, які допускаються у VFS. Цей спосіб описується для кожного типу ФС у структурі `vnodeops`. Як видно зі складу списку абстрактних операцій, вони утворені об’єднанням операцій, характерних для найбільш поширених ФС UNIX. Для того щоб звернення до специфічних функцій не залежало від типу ФС, для кожної операції у `vnodeops` визначається макрос із загальним для всіх типів ФС іменем, наприклад, *VOP_OPEN*, *VOP_CLOSE*, *VOP_READ* і т. ін. Ці макроси визначаються у файлі й відповідають системним викликам. Таким чином, у структурі `vnodeops` приховані залежні від типу ФС реалізації стандартного набору операцій над файлами. Навіть якщо ФС якого-небудь конкретного типу не підтримує певну операцію над своїми файлами, вона повинна створити відповідну функцію, яка виконує деякий мінімум дій: або відразу повертає успішний код завершення, або повертає код помилки. Для аналізу та оброблення повного імені файла в VFS використовується операція *VOP_LOOKUP*, яка дозволяє за іменем файла знайти посилання на його структуру `vnode`.

Структура vnode. Структура vnode (рис. 14) використовується ядром для зв’язку файла з певним типом ФС через поле `v_vfsp` і конкретними реалізаціями файлових операцій через поле `v_op`. Поле `v_pages` використовується для вказівки на таблицю фізичних сторінок пам’яті у разі, коли файл відображається у фізичну пам’ять. У `vnode` також міститься тип файла та вказівник на залежну від типу ФС частину опису характеристик файла — структуру `inode`, що звичайно містить адресну інформацію про розміщення файла на носії та про права доступу до файла. Окрім цього, `vnode` використовується ядром для зберігання інформації про блокування (`locks`), застосовані процесами до окремих ділянок файла.

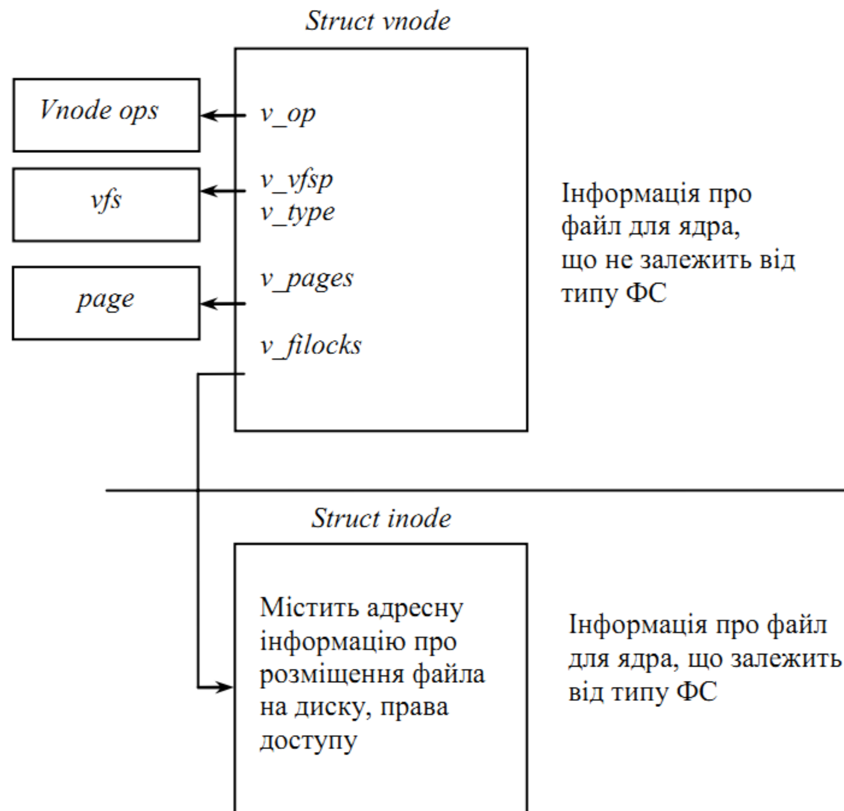

Рисунок 14 – Структура `vnode`

Структура FILE. Із кожним відкриттям процесом файла ядро створює в системній ділянці пам'яті нову структуру типу `file`, яка, як і в разі традиційної ФС s5, описує як відкритий файл, так і операції, які процес збирається проводити з файлом (наприклад, зчитування). Структура `file` містить такі поля:

– `flag` – визначення режиму відкриття (тільки для зчитування, для зчитування й запису і т. ін.);

– `S truct vnode * f_vnode` – вказівник на структуру `vnode` (що замінив порівняно з s5 вказівник на `inode`);

– `offset` – зсув у файлі під час операцій зчитування/записування;

– `struct cred * f_cred` – вказівник на структуру, що містить права процесу, який відкрив файл (структура міститься у дескрипторі процесу);

– вказівники на попередню та подальшу структуру типу `file`, що зв'язують усі такі структури в список.

File і vnode. На відміну від структур типу `file` структури типу `vnode` заводяться ОС для кожного активного (відкритого) файла в єдиному екземплярі, тому структури `file` можуть посилатися на одну й ту ж структуру `vnode`.

Структури `vnode` не зв'язані в який-небудь список. Вони з'являються на вимогу в системному пулі пам'яті й приєднуються до структури даних, яка ініціювала появу цього `vnode`, за допомогою відповідного вказівника. Наприклад, для структури `file` в ній використовується вказівник `f_vnode` на відповідну структуру `vnode`, що описує потрібний файл. Аналогічно, якщо файл пов'язаний з образом процесу (тобто це файл, що містить виконуваний модуль), то сегмент пам'яті, що містить частини цього файла, відображається за допомогою вказівника `vp` (у структурі `segvn_data`) на `vnode` цього файла.

Усі операції з файлами в UNIX System V Release 4 виконуються за допомогою зв'язаної з файлом структури `vnode`. Коли процес запрошує операцію з файлом (наприклад, операцію `open`), то незалежна від типу ФС частина ОС передає керування залежної від типу ФС частини

ОС для виконання операції. Якщо залежна частина виявляє, що структури `vnode`, яка описує потрібний файл, немає в оперативній пам’яті, то залежна частина ствлює для нього нову структуру `vnode`.

Система введення-виведення в операційній системі Unix System V. Підсистема буферизації. Основу системи введення-виведення ОС UNIX складають драйвери зовнішніх пристроїв і засобу буферизації даних. Операційна система UNIX використовує два різні інтерфейси із зовнішніми пристроями: *байт-орієнтований* і *блок-орієнтований*.

Будь-який запит на введення-виведення до блок-орієнтованого пристрою перетвориться в запит до підсистеми буферизації, яка є буферним пулом і комплексом програм керування цим пулом. Буферний пул складається з буферів, що містяться в режимі ядра.

Розмір окремого буфера дорівнює розміру блока даних на диску. З кожним буфером пов’язана спеціальна структура – заголовок буфера, в якому міститься така інформація:

– дані про стан буфера:
- зайнятий/вільний,
- зчитування/записування,
- ознака відкладеного запису,
- помилка введення-виведення;

– дані про пристрій – джерело інформації, що міститься в цьому буфері:
- тип пристрою,
- номер пристрою,
- номер блока на пристрої,
- адреса буфера.

Драйвери. Драйвер – це сукупність програм (секцій), призначена для керування передачею даних між зовнішнім пристроєм і оперативною пам’яттю.

Зв’язок ядра системи з драйверами забезпечується за допомогою двох системних таблиць:

– `bdevsw` – таблиця блок-орієнтованих пристроїв;

– `cdevsw` – таблиця байт-орієнтованих пристроїв.

Для зв’язку використовується така інформація з індексних дескрипторів спеціальних файлів:

– клас пристрою (байт-орієнтований або блок-орієнтований);

– тип пристрою (стрічка, гнучкий диск, жорсткий диск, друкувальний пристрій, дисплей, канал зв’язку і т. ін.);

– номер пристрою.

Висновки.

• Файл – це набір даних, до якого можна звертатися за іменем. Файли поділяються на файли з прямим і послідовним доступом.

• Файлова система – це підсистема ОС, що підтримує логічну і фізичну організації файлів, а також набір системних викликів, що реалізують операції над файлами. Для розміщення структури файлової системи виділяється частина фізичного дискового простору – розділ.

• Для задання декількох імен для одного й того самого файла використовують жорсткі і символьні зв’язки.

• Базові операції для роботи з файлами і каталогами підтримує ОС;

• Головки зчитують інформацію з доріжок. Сукупність усіх доріжок одного радіуса на всіх поверхнях пластин називають циліндром. ОС розподіляє дисковий простір дисковими блоками.

• Розділи диска можуть бути двох типів – первинний і розширений. Тільки один первинний розділ може бути завантажувальним (активним). З активного розділу завантажується програма завантаження завантажувача з якого-небудь іншого розділу, і вже за його допомогою завантажується ОС.

• Оскільки до завантаження ОС система керування файлами працювати не може, то для адресації завантажувачів ОС використовуються абсолютні адреси у форматі Cylinder-Header-

Sector. За фізичною адресою [0-0-1] на диску розміщується головний завантажувальний запис.

• У стандарті формату розміщення таблиці розділів GPT усунуто недоліки існуючих завантажувачів.

• Існують різні підходи до розміщення файлів: неперервний, списковий, індексований.

• Файлова система NTFS спроектована для підвищення надійності ОС Windows.

• В ОС UNIX реалізований механізм віртуальної ФС VFS, який дозволяє ядру системи одночасно підтримувати декілька різних типів ФС.

Література.

1. *Гордеев В.В., Молчанов А.Ю.* Системное программное обеспечение. – СПб.: Питер, 2001. – 736.

2. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.

Запитання.

1. Що таке файлова система і файл?
2. Як організовується інформація у файловій системі?
3. Жорсткі і символьні зв'язки файлів.
4. Операції ОС над файлами і каталогами.
5. Фізична організація файлової системи.
6. Послідовність завантаження ОС.
7. Стандарт розміщення таблиці розділів GPT.
8. Особливості розміщення файлів – неперервного, спискового, індексованого.
9. Структура індексного дескриптора `inode`.
10. Файлова система NTFS.
11. Файлові системи UNIX.

6. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РОЗРОБЛЕННЯ СПЗ

Мета. Вивчення інструментальних засобів розроблення системного програмного забезпечення.

Вступ. Системне програмне забезпечення розробляється з використанням різних інструментальних засобів, до яких відносяться системи керування версіями, інтегровані засоби розроблення, редактори кодів, компілятори, засоби описання компіляції та компонування програм, трасувальники і налагоджувачі для виявлення помилок у бінарних кодах та інші.

План.
1 Системи керування версіями
1.1 Централізована система керування версіями Subversion
1.2 Розподілена система керування версіями Mercurial
1.3 Розподілена система керування версіями Git
2 Інтегровані засоби розроблення та редактори коду
3 Вимоги до сучасного редактора коду
4 Засіб описання компіляції та побудови програм
4.1 Складні командні рядки
4.2 Змінні
4.3 Суфіксні правила
5 Налагодження і трасування програм, виявлення помилок у пам'яті

1 Системи керування версіями

Системи керування/контролю версіями (від англ. Version Control System, VCS, Revision Control System, RCS або Source Code Management, SCM) – програмне забезпечення для полегшення роботи з мінливою інформацією. Система керування версіями дозволяє зберігати декілька версій одного і того ж документа, при необхідності повертатися до більш ранніх версій, визначати, хто і коли зробив ту чи іншу зміну, і багато іншого. Кожна версія позначається унікальною цифрою чи літерою, зміни документу занотовуються. Звичайно також зберігається автор зробленої зміни та її час.

Такі системи найбільш широко використовуються при розробленні програмного забезпечення для зберігання початкових кодів розроблюваної програми. Однак вони також застосовуються і в інших областях, в яких ведеться робота з великою кількістю безперервно змінюваних електронних документів. Зокрема, системи керування версіями застосовуються в САПР, в складі систем управління даними про виріб (PDM), в інструментах конфігураційного управління (Software Configuration Management Tools).

Інструменти для контролю версій входять до складу багатьох інтегрованих середовищ розробки. Системи керування версіями є двох основних типів: з централізованим (Subversion) та розподіленим сховищем (Mercurial, Git).

При роботі з система керування версій використовуються наступні поняття:
• основна гілка коду (trank);
• відгалуження (branch);
• відправлення коду із змінами в репозиторій (submit, commit, check-in);
• одержання коду із змінами з репозиторію (check-out);
• конфлікти (виникають, коли декілька людей корегують один і той же код);
• латка (фрагмент коду із змінами, який записується у сховище, patch).

1.1 Централізована система керування версіями Subversion

Subversion – централізована система керування версіями, в якій дані зберігаються в єдиному сховищі. Сховище може розташовуватися на локальному диску або на мережевому сервері.

Клієнти копіюють файл зі сховища, створюють локальні робочі копії і вносять в них зміни, а потім фіксують ці зміни в сховищі. Декілька клієнтів можуть одночасно звертатися до сховища. Для спільної роботи над файлами в Subversion переважно використовується модель *копіювання – зміна – злиття*. Крім того, для файлів, що не допускають злиття (різні бінарні формати файлів), можна використовувати модель *блокування – зміна – розблокування*.

При збереженні нових версій використовується дельта-компресія: система знаходить відмінності нової версії від попередньої і записує тільки зміни, уникаючи дублювання даних.

Нумерацію всіх версій система робить сама, за командою `svn commit` номер версії всіх файлів проекту збільшується на 1.

Структура каталогів системи:

• `trunk` – поточна робоча версія, що знаходиться у роботі. На відміну від завершеної версії (release) вона не стабільна, тобто розробляється, компілюєтсья і запускається, не заважаючи іншим розробникам тестувати власний код.

• `branch` – відгалуження від проекту. Воно використовується при необхідності вдосконалити версію, яка пишеться в `trunk`, але саме вдосконалення є складним і його реалізація може завадити іншим працювати. В деяких випадках коли реалізація певних частин програми займає багато часу, також створюють `branch`; у таких випадках назву `branch` дають таку ж як назва модуля, що розробляється. Як тільки розроблення модуля буде завершено, викликається процес добавлення `branch` в `trunk`.

• `tags` – часові позначки для `trunk`, які використовуються в нумерації завершених версій.

При використанні доступу за допомогою WebDAV також підтримується прозоре управління версіями – якщо будь-який клієнт WebDAV відкриває для запису і потім зберігає файл, що знаходиться на мережевому ресурсі, то автоматично створюється нова версія.

Переваги Subversion**:**

• графічні інтерфейси і підтримка роботи з консолі;

• відслідковується історія зміни файлів і каталогів навіть після їх перейменування і переміщення;

• висока ефективність роботи з текстовими і двійковими файлами;

• вбудована підтримка інтегрованих середовищ розроблення (KDevelop, Zend Studio та інших);

• можливість створення дзеркальних копій репозиторію;

• два типи репозиторію – база даних або набір звичайних файлів;

• можливість доступу до репозиторію через Apache з використанням протоколу WebDAV;

• наявність зручного механізму створення позначок і гілок проектів.

Недоліки Subversion**:**

• повна копія репозиторію зберігається на локальному комп'ютері у прихованих файлах, що потребує достатньо великого обсягу пам'яті;

• слабо підтримується операція об'єднання гілок проекту;

• складність з повним вилученням інформації про файли, які попали у репозиторій.

1.2 Розподілена система керування версіями Mercurial

Mercurial – вільна розподілена система керування версіями файлів та спільної роботи з дуже великими репозиторіями початкового коду. Mercurial є консольною програмою, написаною на мові Python. Найбільш критичні ділянки коду написані на мові C.

Для ідентифікації версій використовується алгоритм хешування SHA1 (Secure Hash Algorithm 1), але також передбачена можливість присвоєння індивідуальних номерів. Підтримується можливість створення гілок проекту з подальшим їх об'єднанням.

Для взаємодії між клієнтами використовуються протоколи HTTP, HTTPS або SSH.

Набір команд простий і зрозумілий, подібний до команд Subversion. Є графічні оболонки і доступ до репозиторію через веб-інтерфейс, а також утиліти імпорту репозиторію у інші системи контролю версій.

Основні переваги Mercurial:

• швидка обробка даних;

• платформо-незалежна підтримка;

• можливість роботи з декількома гілками проекту;

• простота у використанні;

• можливість конвертування репозиторіїв інших систем підтримки версій (CVS, Subversion, Git, Darcs, GNU Arch, Bazaar та інші).

Недоліки Mercurial**:**

• можливість (надзвичайно мала) співпадіння хеш-коду різних за змістом версій;

• орієнтація на роботу в консолі.

1.3 Розподілена система керування версіями Git

Git – розподілена система (без єдиного сервера) керування версіями файлів та спільної роботи. Проект створив Лінус Торвальдс для керування розробкою ядра Linux. Git є однією з найефективніших, надійних і високопродуктивних систем керування версіями, що надає гнучкі засоби нелінійної розробки, що базуються на відгалуженні і злитті гілок. Для забезпечення цілісності історії та стійкості до змін заднім числом використовуються криптографічні методи, також можлива прив'язка цифрових підписів розробників до тегів і записів у репозиторій.

Git надає безліч можливостей не тільки розробникам програм, але і письменникам для змін, доповнень і відслідковування «рукописів» і сюжетних ліній, і вчителям для корегування і розвитку курсу лекцій, і адміністраторам для ведення документації, і для багатьох інших напрямків, в яких вимагається керування історією змін.

Кожний розробник, який використовує Git, має свій локальний репозиторій, який дозволяє локально керувати версіями. Потім, збереженими в локальному репозиторії даними, можна обмінюватися з іншими користувачами.

Часто при работі з Git створюють центральний репозиторій, з яким інші розробники синхронізуються. Приклад організації системи с центральним репозиторієм є проект розроблення ядра Linux (`http://www.kernel.org`). У цьому випадку всі учасники проекту ведуть свої локальні розробки і безперешкодно скачують оновлення з центрального репозиторію. Коли необхідні роботи окремими учасниками проекту виконані і налагоджені, вони, після посвідчення власником центрального репозиторію у коректності і актуальності виконаної роботи, завантажують свої зміни у центральний репозиторій. Наявність локальних репозиторіїв також суттєво підвищує надійність зберігання даних.

Робота над версіями проекту в Git може вестися у декількох гілках, які потім можна повністю або частково об'єднати, знищити, відкотити і розгалузити у нові гілки проекту.

Переваги Git:

• надійна система порівняння версій і перевірки коректності даних, основана на алгоритмі хешування SHA1 (Secure Hash Algorithm 1);

• гнучка система галуження проектів і злиття гілок між собою.

• наявність локального репозиторію, який містить повну інформацію про всі зміни і дозволяє здійснювати локальний контроль версій і записувати у головний репозиторій тільки повністю перевірені зміни;

• висока продуктивність і швидкість роботи;

• зручний і зрозумілий набір команд;

• набір графічних оболонок для швидкої і якісної роботи із системою;

• можливість робити контрольні точки, в яких дані зберігаються без дельта компресії, а повністю. Це дозволяє збільшити швидкість відновлення даних, так як за основу береться найближча контрольна точка;

• широка поширеність, доступність і якісна документація;

• гнучкість системи дозволяє її зручно налаштувати і навіть створювати спеціалізовані системи контролю або інтерфейси користувача бази;

• універсальний мережевий доступ з використанням протоколів `http, ftp, rsync, ssh` і інших;

Недоліки Git:

• Unix/Linux орієнтованість;

• можливість (надзвичайно мала) співпадіння хеш-коду різних за змістом версій;

• не відслідковуються зміни окремих файлів, а тільки всього проекту загалом, що може бути незручним при роботі з великими проектами, які містять множину незв’язаних файлів;

• при початковому (першому) створенні репозиторію і синхронізації його з іншими розробниками, потрібний достатньо тривалий час для скачування даних, особливо, якщо проект великий, так як потрібно скопіювати на локальний комп’ютер увесь репозиторій.

2 Інтегровані середовища розроблення та редактори коду

Інтегровані середовища (засоби) розроблення (IDE) не є критично необхідним компонентом розроблення програм. У традиціях Unix/Linux цілком достатнім для розроблення програм є використання текстового редактора, який має додаткові властивості, такі як кольорова розмітка тексту, функції контекстного пошуку і заміни. Таких редакторів в Linux достатньо, починаючи з vim, kword, Kate, Emacs і до простого редагування в `mc` клавішею F4. Практика показує, що цих засобів цілком достатньо аж до середніх розмірів проектів.

Але використання IDE часто дозволяє більш продуктивно опрацювати програмний код, оперативніше виконати в зв’язці цикл: редагування коду – складання проекту – налагодження.

Інтегровані середовища розроблення можуть містити наступні елементи:

• текстовий редактор;

• компілятор і/або інтерпретатор;

• засоби автоматизації компонування;

• налагоджувач;

• конструктор графічного інтерфейсу;

• оглядач класів, інспектор об’єктів, діаграму ієрархії класів (при ООП);

• система керування версіями.

В Linux доступні IDE з різним ступенем інтегрованості. Нижче розглянуто найбільш відомі IDE.

Eclipse IDE (Eclipse Integrated Development Environment, `http://www.eclipse.org`) – один з найбільш відомих на сьогодні засобів розроблення. Активно розвивається з 2000 р., спочатку як пропрієтарний проект IBM, який потім був перетворений у відкритий проект. Відмінною особливістю є можливість динамічних розширень (які може підготувати і звичайний користувач), за рахунок цього напрацьовані плагіни для різних мов програмування (Java, Ci/C++, PHP, Python, Ruby, Ada і багатьох інших). Eclipse IDE опрацьований практично для всіх операційних систем, так як реалізований на мові Java. Eclipse IDE є багато-платформовим середовищем не тільки для ОС, але і для безлічі апаратних платформ відмінних від Intel x86, для яких може вестися крос-розроблення: ARM, MIPS, PPS і навіть мікроконтролери, наприклад, AVR. Окрім засобів розроблення в Eclipse IDE включаються, як динамічні додатки (додатки, що динамічно підключаються до основної програми з метою розширення її можливостей, plug-in) програмні емулятори інших апаратних платформ (наприклад, Android ARM) з метою налагодження. На основі Eclipse IDE сторонніми розробниками створено багато інших IDE, спеціалізованих під конкретні застосування, і це створює складності у виборі конкретної модифікації IDE.

Eclipse IDE наявний в репозиторіях практично будь-якого дистрибутива Linux, звідки може бути встановлений.

Так як IDE є безкоштовним і має високу якість, то в багатьох організаціях прийнятий за корпоративний стандарт для розроблення застосувань.

KDevelop (`www.kdevelop.org`) – вільне середовище розроблення програмного забезпечення для Linux, Solaris, FreeBSD, Mac OS X, Windows і різних Unix-систем, яке засноване на бібліотеках KDE/Qt і повністю підтримує процес розроблення для KDE.

Проект стартував в 1998 році. KDevelop розповсюджується згідно з GNU General Public License. KDevelop не має свого компілятора, а використовує GNU Compiler Collection (або будь-який інший компілятор) для створення виконуваного коду. Основною мовою розроблення є C++, але з використанням плагінів забезпечується підтримка додаткових мов програмування, (Сі, Java, PHP, Ruby, Python, Ada, Bash, Fortran, Pascal, SQL, Perl і Bash). Крім того, доступні плагіни для інтеграції з інструментами Valgrind, QTest, qmake, Mercurial і Perforce (Subversion і Git підтримуються штатно).

KDevelop не залежить від мови програмування і не залежить від платформи, на якій він запускається, підтримуючи KDE, GNOME і багато інших технологій (наприклад, Qt, GTK+ і wxWidgets). Підтримуються такі утиліти, як GNU (`automake`), `qmake` і `make` для власних засобів складання проектів. Доповнення коду доступно для мов Сі і C++. Символи зберігаються в Berkeley DB файлі для швидкого пошуку без попереднього розбору.

Kdevelop вміє генерувати початкові скелети додатків. Відмінною особливістю Kdevelop (великим плюсом в деяких випадках) є те, що серед таких шаблонів є і проект модуля ядра (драйвера) Linux.

MonoDevelop (`www.monodevelop.com`) – відкрите інтегроване середовище розроблення програм для платформ Linux, Mac OS X та Microsoft Windows з використанням Mono і Microsoft.NET framework. На даний момент підтримуються мови C#, Java, Boo, Visual Basic.NET, CIL, Python, Vala, C та C++. Також MonoDevelop підтримує такі технології, як Gtk#, ASP.NET MVC, Silverlight, MonoMac и MonoTouch.

MonoDevelop включає можливості подібні до NetBeans та Microsoft Visual Studio, такі як автоматичне доповнення, інтеграція контролю коду, графічний інтерфейс користувача і веб-дизайнер. В MonoDevelop інтегрований Gtk# GUI дизайнер під назвою Stetic.

Qt Creator – інтегроване вільне середовище для розроблення програм мовами C, C++ і QML. IDE підтримується компанією Digia в рамках технології Qt. Включає в себе графічний інтерфейс налагоджувача і візуальні засоби розроблення графічного інтерфейсу як з використанням QtWidgets, так і QML. Підтримує компілятори: GCC, Clang, MinGW, MSVC, Linux ICC, GCCE, RVCT, WINSCW.

Anjuta (`http://www.anjuta.org`) – інтегроване середовище розроблення GNOME для мов C, C++, Vala, Java, JavaScript, Python. Особливо добре підходить для розроблення графічних програм. Середовище містить: засоби керування проектом, майстри застосувань, вбудований інтерактивний налагоджувач, редактор початкового коду із засобами перегляду і підсвічування синтаксису, систему контролю версій, конструктор графічного інтерфейсу.

Geany (http://www.geany.org) – популярний серед багатьох розробників, простий в роботі багато-платформовий інструмент. Geany не має власного компілятора, а використовує компілятори з GNU колекції і або будь-який інший. По суті, Geany не є IDE, а є інструмент редагування кодів з розміткою кольором та вбудованим викликом `gcc`, `маке`, `mc`. Завдяки такій специфіці Geany використовується при розробленні програм на різних мовах програмування, серед яких Сі, C++, Java, JavaScript, Tcl, PHP, Python, XML/HTML та інші.

В Geany реалізовані наступні функції:

• підсвічування сирцевого коду з урахуванням синтаксису мови програмування;
• автозавершення слів;
• автоматичне підставлення функцій (стандартних і з відкритих файлів);
• простий менеджер проектів;
• підтримка динамічних додатків;
• вбудований емулятор термінала.

• налагодження коду за допомогою GDB.

Однак, впровадження нових мов програмування в інтегровані середовища є складною і трудомісткою задачею. Крім того, IDE проекти є великі за обсягом, так як вони містять багато допоміжних файлів Тому альтернативою інтегрованим середовищам розроблення є редактори кодів.

3 Вимоги до сучасного редактора коду

Текстові редактори відрізняються від текстових процесорів своїм призначенням. Текстові редактори призначені для читання і зміни довільного текстового файлу, тому їх функції зосереджені навколо маніпуляцій з текстом. Текстові процесори потрібні для створення та форматування текстових документів.

При написанні програмного коду використовуються текстові редактори з додатковими функціями, основними з яких є підсвічування синтаксису і автоматичні відступи. Підсвічування синтаксису яскраво виділяє ключові слова, а також показує різними кольорами імена змінних і дані. Крім того, кольорами можуть виділятися змінні різних типів або виклики функцій. Автоматичні відступи допомагають побачити кожен блок коду окремо, вкладені блоки розміщуються із наростаючими відступами, що істотно спрощує читання.

Більшість редакторів забезпечують автоматичне доповнення коду для різних мов програмування, таких як Pascal, C/C++, C#, Java, Ruby, Python, PHP, Асемблер, HTML, JavaScript та інші. Під час введення даних на екрані з'являється список можливих підстановок для доповнення введеного коду. Введення перших літер у слові певного формату активує автодоповнення слова. Редактори також забезпечують перехід до будь-якого файлу, типу або символу за допомогою визначеної комбінації клавіш.

Сучасний редактор програмного коду повинен підтримувати наступні можливості:

- підсвічування тексту і можливість згортання блоків, згідно синтаксису мови програмування;
- підтримка різних мов (C, C++, C#, Java, Python, Ruby, HTML, PHP, Java Script, SQL, CSS, Pascal, Perl, Bush, Lua, TCL, Assembler);
- WYSIWYG (друкування і отримування того, що видно на екрані);
- налагодження користувачем режимів підсвічування синтаксису;
- авто-завершення слова, що набирається.
- одночасна робота з багатьма файлами;
- одночасний перегляд декількох файлів;
- підтримка регулярних виразів "пошук/заміна".
- підтримка перетягування фрагментів тексту:
- автоматичне визначення стану файлу (збережений, не збережений);
- збільшення і зменшення масштабу;
- нотатки (написання коментарів);
- виділення дужок при редагуванні тексту.

Такі можливості забезпечують сучасні універсальні редактори коду Emacs, VIM, jEdit, Notepad++, Kate. Їх описи синтаксисів та аналіз структури реалізовані найбільш широко і повно. Вбудовані в них синтаксичні аналізатори на основі внутрішніх описів граматики мов виконують роботу з розбору тексту і надають на основі цієї інформації розширені сервіси.

4 Засіб описання компіляції та побудови програм

Одним із засобів описання компіляції та побудови програм є `make`. `Make` дозволяє однією командою скомпілювати і побудувати програму, яка складається з багатьох модулів. Більш того, якщо є множина файлів для компіляції, а код був змінений тільки в деяких з них, то `make` створить об'єктні файли тільки для змінених файлів. Для того щоб `make` виконав таку роботу потрібно описати усі файли у файлі з іменем `makefile`. Приклад такого файлу:

```
1 #makefile
2 OBJS = main.o sub1.o sub2.o
3 LDLIBS = -L/usr/local/lib/ -lbar
4 main: $(OBJS)
5     g++ - o main $(OBJS) $(LDLIBS)
6 install: main
7      install –m 644 main /usr/bin
8 .PHONY: install
```

Рядок 1 – це коментар.

В рядку 2 визначається змінна з іменем `OBJS` як `main.o sub1.o sub2.o`.

В рядку 3 визначається інша змінна з іменем `LDLIBS`.

В рядку 4 починається визначення *правила*, яке вказує на те, що файл `main` залежить від файлів, які містяться у змінній `OBJS`. Файл `main` називається цільовим об'єктом, а `$(OBJS)` – списком залежностей. Синтаксис розширення змінної: ім'я змінної поміщається в `$(...)`.

Рядок 5 вказує на те, як побудувати цільовий об'єкт із списку залежностей.

Рядок 6 вказує, що потрібно інсталювати файл `main` за допомогою програми `install`.

Рядок 7 інсталює отриманий бінарний файл `main` в каталог `/usr/bin` за допомогою стандартної програми `install`.

Рядок 8 використовує директиву `.PHONY`, яка змінює операцію `make`. Вона вказує `make` на те, що цільовий об'єкт `install` не є іменем файлу. Цільові об'єкти `.PHONY` часто використовуються для інсталяції або створення одиночного імені цільового об'єкта, який основується на декількох інших уже існуючих цільових об'єктах.

Аргумент `–k` заставляє `make` створювати максимально можливу кількість файлів без зупинки, навіть якщо якась із команд повернула помилку.

Якщо відомо, що якась команда буде завжди повертати помилку, а її потрібно проігнорувати, то можна скористатися командами оболонки. Команд `/bin/false` завжди припиняє роботу, якщо тільки не вказана опція `-k`. Конструкція `люба_команда || /bin/true` ніколи не перериває роботу, навіть якщо люба_команда повертає `false`.

При запуску команди `make`, вона читає і порядково виконує вміст файлу `makefile`, компілюючи описані програми, підключаючи об'єктні файли із бібліотек і створює в кінці бінарний файл. Якщо люба із заданих команд дає помилку `make` припиняє роботу.

4.1 Складні командні рядки

Кожний командний рядок виконується у своїй власній підоболонці. Таким чином, команди `cd` в командному рядку впливають тільки на рядок, в якому вони записані. Любий рядок в make-файлі можна розширити на множину рядків, вказаних в кінці символом "\". Приклад, як можуть виглядати командні рядки:

```
1 cd перший_каталог; \
2    зробити щось з файлом $(FOO); \
3    зробити ще щось
4 cd другий каталог;  \
5     if [ -f деякий файл ] ; then
6          зробити що-небудь інше;   \
7      done;  \
8  for i in * ; do \
9           echo $$i >> деякий файл; \
10  do
```

`Make` знаходить у цьому фрагменті коду тільки два рядки. Перший командний рядок починається з 1 і закінчується в 3, а другий починається з рядка 4 і закінчується в рядку 10. Деякі зауваження до коду:

• другий каталог є відносним не до каталогу `перший_каталог`, а до каталогу в якому запущений `make`, так як ці команди виконуються у різних оболонках;

• стрічки, які утворюють кожний командний рядок, передаються оболонці як один рядок. Тому всі символи `;`, які потрібні оболонці, необхідно задати, навіть ті, які звичайно у сценаріях пропускаються;

• якщо потрібно розіменувати змінну `make`, то використовують запис `$(змінна)`, а якщо змінну оболонки - то `$$i`.

4.2 Змінні

Для визначення тільки одного компоненту змінної за один раз, можна записати так:

```
OBJS = foo.o
OBJS = $(OBJS) bar.o
OBJS = $(OBJS) raz.o
```

Очікується, що `OBJS` буде визначена як `foo.o, bar.o, raz.o`, а в дійсності вона буде визначена як `$(OBJS) raz.o`. При посиланні в правилі на `OBJS` `make` ввійде у нескінченний цикл. Тому `make` розділи задають наступним чином:

```
OBJS1 = foo.o
OBJS2 = bar.o
OBJS3 = baz.o
OBJS = $(OBJS1) $(OBJS2) $(OBJS3)
```

Існує форма *простого присвоєння змінних*:

```
OBJS := foo.o
OBJS := $(OBJS) bar.o
OBJS := $(OBJS) raz.o
```

Операція := заставляє GNU обчислити вирази змінної при присвоєнні, а не чекати обчислення виразу при його виконанні у правилі. В результаті виконання цього коду OBJS дійсно отримає `foo.o bar.o raz.o`.

Існує і аналогічний інший синтаксис присвоєння:

```
OBJS := foo.o
OBJS += $(OBJS) bar.o
OBJS += $(OBJS) raz.o
```

4.3 Суфіксні правила

Суфіксні правила виглядають наступним чином:

```
.c.o:
    $(CC) -c (CFLAGS) $(CPPFLAGS) -o $@ $<
.SUFFIXES: .c .o
```

Це правило вказує, що `make` повинно перетворити файл `a.c` в `a.o` шляхом запуску вказаного командного рядка. У цьому правилі використовуються автоматичні змінні. Автоматична змінна `$@` виступає як цільовий об'єкт, а `@<` – як перша залежність, `$^` – остання залежність. Існують і інші автоматичні змінні, які є довідках по `make`. Всі автоматичні змінні можна використовувати у звичайних, суфіксних і шаблонних правилах.

Останній рядок прикладу `.SUFFIXES` вказує `make` на те, що `.c` і `.o` є суфіксами, які має використати `make` для знаходження способу перетворити існуючі початкові файли у потрібні цільові об'єкти.

Шаблонні правила більш потужніші, а отже і більш складніші ніж суфіксні правила. Приклад еквівалентного шаблонного правила для показаного вище суфіксного правила:

```
%.o %.c
    $(CC) -c (CFLAGS) $(CPPFLAGS) -o $@ $<
```

Більшість великих проектів з відкритим початковим кодом використовують інструменти `Automake`, `Autoconf`, `Libtool`. `Automake` пише цільові об'єкти `install` і `uninstall`, `Autoconf` автоматично визначає можливості системи і налаштовує програмне забезпечення для його відповідності системі, а `Libtool` відслідковує відмінності у керуванні сумісно використовуваними бібліотеками на різних системах.

5 Налагодження і трасування програм, виявлення помилок в пам'яті

Для *покрокового виконання, аналізу і виявлення помилок у виконуваному файлі* використовується налагоджував (debugger). При розробленні вільного програмного забезпечення в середовищі Linux використовується налагоджувач консольного рядка **gdb**.

Налагоджуваний виконуваний файл має містити додаткову символьну інформацію. Для її добавлення потрібно при компіляції і компонуванні виконуваного файлу вказати додаткові ключі, наприклад

```
>gcc -g prog.c –o prog
>g++ -g prog.cpp –o prog
>gcc -ggdb -g3 prog.cpp –o prog
```

Налагоджувач запускається з консолі командою:

```
>gdb імя_виконуваного_файлу
```

`Gdb` не буде продивлятися значення `PATH` при пошуку виконуваного файлу. `Gdb` завантажить символьну інформацію для виконуваного і видасть запит на подальші дії.

Існує три способи перевірити процес за допомогою `gdb`:

• використовуючи команду `run` для звичайного виконання програми;

• використовуючи команду `attach` для початку перевірки уже запущеного процесу. При підключенні до процесу останній зупиняється;

• використовуючи існуючий файл ядра (core file) для визначення стану процесу при його аварійному завершенні. Для дослідження файлу ядра потрібно запустити команду `gdb core`.

Перед запуском програми або підключенням до уже запущеного процесу можна встановити точку переривання, продивитися початковий код і виконати інші операції, які не обов'язково відносяться до запущеного процесу.

`Gdb` не вимагає введення повного імені команди, наприклад для `run` достатньо вказати `r`, для `next` – `n`, для `step` – `s`.

`Gdp` підтримує оперативну довідку, яку можна отримати ввівши команду `help` або `help команда`, `help тема`.

Деякі команди оболонки `gdb` сприймають ідентифікатори формату для специфікації виведення значень. Ідентифікатори формату розміщуються за іменем команди, відділяються від неї символом "/" і складаються з трьох елементів: цифра, буква формату і буква розміру (необов'язкові).

Букви формату можуть мати наступні значення: `o` – вісімкове число, `x` – шістнадцяткове число, `d` – десяткове число, `u` – без знакове десяткове число, `t` – двійкове число, `f` – число з плаваючою крапкою, `a` – адреса, `i` - інструкція, `c` – символ, `s` – стрічка.

Символи задають розмір даних: b – байт, h – півслово (2 байти), w – слово (4 байти), g – чотирне слово (8 байтів).

Перелік найбільш вживаних команд gdb:

`attach, at`	Підключає налагоджував до вже запущеного процесу. Єдиним аргументом є ідентифікатор процесу (pid), до якого здійснюється підключення. Процеси, з якими встановлено підключення, зупиняються, перериваючи любі очікуючі або поточні системні виклики, які дозволено переривати.
`backtrace, bt,`	виводить трасування стеку

where, w	
break, b	Встановлює точку переривання. Можна вказати ім'я функції, номер рядка поточного файлу, пару імя_файлу:номер_рядка, довільну адресу *адреса. Gdb назначає і виводить унікальний номер для кожної точки переривання.
clear	Вилучає точку переривання визначену номером.
condition	Змінює точку переривання, визначену номером для переривання, тільки якщо вираз має значення істина.
continue, c	Продовження виконання програми до наступної точки переривання (breakpoint).
delete	Вилучає точку переривання визначену номером.
detach	Від'єднання від поточного підключеного процесу.
display	Відображає значення виразу кожний раз при зупинці виконання. Приймає такі ж аргументи (включно з модифікаторами формату), як print. Виводить номер виведення, який надалі може використовуватися для відміни виведення.
help	Викликає довідку
jump	Переходить на задану адресу і продовжує виконання процесу з цієї адреси. Адресу можна задати як номер рядка або як *адрес.
list, l	Без аргументів виводить 10 рядків оточуючих поточну адресу. Наступні виклики list виводять наступні 10 рядків. При використанні аргументу ″-″ (list -) виводить 10 попередніх рядків. З аргументами: Файл:номер_рядка, номер_рядка – виводить 10 оточуючих рядків; Файл:функція, функція – виводить 10 оточуючих рядків; *адреса - виводить 10 оточуючих рядків.
next, t	Переходить на наступний рядок початкового коду в поточній функції, без заходження всередину функції.
nexti	Переходить на наступну інструкцію машинного коду без заходження всередину інструкції.
print, p	Виводить значення у зрозумілій формі. Якщо є змінна char* c, то команда print c виведе адресу стрічки, а print *c виведе саму стрічку. Для структур виводяться їх члени. Можна використовувати перетворення типів, які буде враховувати gdb. Якщо код скомпільований з опцією –ggdb, то у виразах стануть доступними перечислимі значення і визначення препроцесора. Команда приймає ідентифікатори формату.
run, r args	Запускає поточну програму спочатку. Аргументи команди run передаються в командний рядок для запуску програми. В gdb можна універсалізувати імена файлів за допомогою * і [], а також здійснювати переадресацію з використанням <, >>, але не можна створювати канали або внутрішні документи. Без аргументів run використовує аргументи, які були визначені в самій останній команді run або set args. Для запуску без аргументів після їх застосування використовується команди set args без додаткових аргументів.
set, s variable=value	Присвоєння значення змінній, наприклад set a=argv[5]. Кожний раз при виведенні виразу за допомогою print створюється змінна виду $1, на яку в подальшому можна посилатися, наприклад set a=$1. Команда set має багато підкоманд, інформацію про які можна отримати командою help set.

`step, s`	Виконує інструкцію програми до нового рядка початкового коду
`stepi`	Виконує одну інструкцію машинної мови, з заходженням усередину інструкції.
`undisplay`	Без аргументів відміняє всі виведення. Інакше відміняє виведення вказані номерами.
`whatis`	Виводить тип даних виразу, переданого як аргумент команди.
`where,w`	Виводить трасування стеку
`x`	Подібна до команди `print` за тим винятком, що явно обмежується виведенням вмісту за вказаною адресою у довільному форматі.

Приклад налагодження Сі програми в GDB:

```
#include<stdio.h>
main() {
    int count;

    for (count=0;count<10;count++)
       printf("Hello from CETS!\n");
}
```

```
>gdb ./myprog
..
Reading symbols from myprog...done.
(gdb) b main
Breakpoint 1 at 0x400568: file 2.c, line 6.
(gdb) r
Starting program: myprog
Breakpoint 1, main () at 2.c:6
6            for (count=0;count<10;count++)
(gdb) s
7               printf("Hello from main!\n");
(gdb) p count
$1 = 0
(gdb) disp count
1: count = 0
(gdb) set count=8
(gdb) s
Hello from main!
6            for (count=0;count<10;count++)
1: count = 8
(gdb)
7               printf("Hello from main!\n");
1: count = 9
(gdb) c
Continuing.
Hello from main!
[Inferior 1 (process 1582) exited with code 021]
(gdb) q
```

Для *трасування виконуваного файлу* використовуються утиліти **`strace`** і **`ltrace`**. Подібно до `gdb, strace i ltrace` можна використовувати для виконання програми від початку до кінця або підключитися до вже запущеної програми. Утиліти мають подібний набір опцій:

`-c` – підрахунок часу, кількості системних викликів;

`-f` – трасування дочірніх процесів.

Утиліта `strace` виводить запис про кожний системний виклик програми. Запуск `strace` на виконання:

```
>strace –опція ./myprog
```

Утиліта `ltrace` виводить запис про кожну функцію бібліотеки, яку викликає програма. Запуск `ltrace` на виконання:

```
>ltrace –опція ./myprog
```

Приклади виконання програм `strace` і `ltrace` з опцією `–c`:

```
>strace –c ./myprog
```

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.000024           2        12         7 open
  0.00    0.000000           0         4           read
  0.00    0.000000           0        10           write
  0.00    0.000000           0         5           close
  0.00    0.000000           0         4         3 stat
  0.00    0.000000           0         6           fstat
  0.00    0.000000           0        17           mmap
  0.00    0.000000           0        10           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           brk
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000024                    73        11 total
```

```
>ltrace –c ./myprog
```

```
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 79.46    0.003165         316        10
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
 10.65    0.000424         424         1 _ZNSt8ios_base4InitC1Ev
  5.90    0.000235         235         1 __cxa_atexit
  3.99    0.000159         159         1 _ZNSt8ios_base4InitD1Ev
------ ----------- ----------- --------- --------------------
100.00    0.003983                    13 total
```

Для *виявлення помилок в динамічній пам'яті* використовується **`valgrind`**. `Valgrind` є спеціальним інструментом, який емулює процесор класу x86 для безпосереднього спостереження за всіма зверненнями до пам'яті і аналізу потоку даних.

Запуск `valgrind` на виконання:

```
>valgrind ./myprog
```

`Valgrind` має опцію, яка дозволяє включити перевірку витікання пам'яті, при якій для кожного виділення знаходяться всі доступні вказівники, які посилаються на цю пам'ять.

```
>valgrind --leak-chek=full ./myprog
```

Приклад програми, яка містить помилку звільнення динамічної пам'яті.

```
include <iostream>
using namespace std;

class A {
   int a[3];
public:
  A(int *a1) { for(int i=0;i<3;i++) a[i]=*(a1+i); }
  void Get(void) { cout << a[0] <<a[1] <<a[2]<< endl; }
};

int main() {
  int b[3]={1,2,3};
```

```
  A obj= A(b);
  obj.Get();

  A *p = new A(b);
  p->Get();
  //delete p;  // Помилка! Не звільнено динамічну пам’ять
  return 0;
}
```

В результаті перевірки `valgrind` виявив, що в динамічній області виділено 1 блок, а звільнено 0.

```
> valgrind --leak-check=full ./myprog
==2182== Memcheck, a memory error detector
==2182== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==2182== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==2182== Command: ./myprog
==2182==
123
123
==2182==
==2182== HEAP SUMMARY:
==2182==     in use at exit: 12 bytes in 1 blocks
==2182==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==2182==
==2182== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2182==    at 0x4C27D49: operator new(unsigned long) (in
/usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==2182==    by 0x400A16: main (in myprog)
==2182==
==2182== LEAK SUMMARY:
==2182==    definitely lost: 12 bytes in 1 blocks
==2182==    indirectly lost: 0 bytes in 0 blocks
==2182==      possibly lost: 0 bytes in 0 blocks
==2182==    still reachable: 0 bytes in 0 blocks
==2182==         suppressed: 0 bytes in 0 blocks
==2182==
==2182== For counts of detected and suppressed errors, rerun with: -v
==2182== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

Висновки.

• Системи керування версіями полегшують роботу з мінливою інформацією. Найбільше поширення отримали системи керування версіями з централізованим (Subversion) та розподіленим сховищем (Mercurial, Git).

• Інтегровані середовища розроблення програм дозволяють більш продуктивно опрацювати програмний код, оперативніше виконати в зв’язці цикл: редагування коду – складання проекту – налагодження;

• Сучасні універсальні редактори Emacs, VIM, jEdit, Notepad++, Kate мають усі необхідні функції для опрацювання програмних сирцевих кодів і тому можуть бути альтернативою до інтегрованих середовищ.

• Одним із поширених засобів описання компіляції та побудови багатомодульних програм є утиліта `make`.

• Для покрокового аналізу і виявлення помилок у виконуваному файлі використовується налагоджувач (debugger).

• Для трасування виконуваного файлу використовуються утиліти `strace` і `ltrace`. Утиліта `strace` виводить запис про кожний системний виклик програми. Утиліта `ltrace` виводить запис про кожну функцію бібліотеки, яку викликає програма. Програма `valgrind` виявляє помилки в динамічній пам’яті.

Література.

1. Блум Ричард, Бреснахэн Кристина. Командная строчка Linux и сценарии оболочки. Библия пользователя, 2-е узд.: Пер. с англ. – М.: ООО "И.Д. Вильямс", 2012. – 784 с.

2. Джонсон, Майкл К., Троан, Єрик В. Разработка приложений в среде Linux, 2-е изд.: Пер. с англ. –М.: "ООО И.Д. Вильямс", 2007. – 544 с.

Запитання.

1. Призначення і області застосування систем контролю версій.
2. Порівняльна характеристика систем контролю версій Subversion, Mercurial, Git.
3. Призначення і можливості інтегрованих середовищ розроблення програм.
4. Порівняльна характеристика інтегрованих середовищ Eclipse, KDevelop, MonoDevelop, QtCreator, Anjuta.
5. Можливості сучасних редакторів програмного коду.
6. Засіб описання компіляції та побудови програм `make`.
7. Складні командні рядки і змінні `make`. Суфіксні і шаблонні правила `make`.
8. Утиліти налагодження і трасування програм `strace`, `ltrace`. Програма виявлення помилок у динамічній пам'яті `valgrind`.

7. КОМАНДИ ОС LINUX

Мета. Вивчення основних груп команд ОС Linux

Вступ. Команди ОС використовуються для роботи з системою з консолі. Команди це програми, які можуть бути як простими, так і складними. Так проста команда `ls` виводить вміст поточного каталогу, а складна команда `gcc` викликає компілятор. Команди, як правило мають аргументи і перемикачі (прапори, опції), які визначають специфіку виконання програми.

План.
1 Команди ОС Linux і їх групи
1.1 Команди Linux впорядковані за алфавітним порядком
2 Системна інформація
3 Файли і каталоги
4 Пошук файлів
5 Перегляд і редагування файлів
6 Архівування і стискання файлів
7 Дисковий простір
8 Встановлення програмних пакетів в SUSE
9 Встановлення програмних пакетів в Fedora, RedHat (YUM)
10 Встановлення програмних пакетів в Debian, Ubuntu (DEB)
11 Перетворення наборів символів і файлових форматів
12 Аналіз файлових систем
13 Створення файлових систем
14 Монтування файлових систем
15 Створення резервних копій (backup)
16 Робота з CDROM
17 Використання ресурсів і пристроїв
18 Мережа (LAN і WiFi)
19 Microsoft Windows networks(SAMBA)
20 Керування міжмережевим екраном IPTABLES (firewall)
21 Моніторинг і налагодження
22 Каталоги файлової системи Linux

1 Команди ОС Linux і їх групи

Команди ОС Linux використовуються у наступних випадках:
- у всіх випадках коли не спрацьовують функції графічного інтерфейсу;
- при віддаленому адмініструванні серверів;
- для реалізації функцій, які не забезпечує графічне середовище;
- якщо не стартує або пошкоджене графічне середовище X Window.

Довідкова інформація про команди Linux зберігається у man сторінках (manual pages), info документах і help повідомленнях. Man сторінки організовані секціями: 0 – заголовкові файли (/usr/include), 1 – виконувані програми і shell команди, 2 – системні виклик, 3 – бібліотечні виклики, 4 – спеціальні файли, 5 – формати файлів і домовленості, 6 – ігри, 7 – різне, 8 – команди системного адміністратора, 9 – підпрограми ядра (не стандартні).

Отримання довідкової інформації про команди:

```
help, man –h     довідкова система man
help, man –h     довідкова система help
mount –h        показати опції команди mount
man mount       знайти man сторінки з описанням команди mount
man hier        виведення на екран описання ієрархії файлової системи Linux
```

info mount показати інформацію про команду mount
apropos mount знайти сторінки в базі man сторінок, які містять слово mount
what_is mount знайти сторінки в базі man сторінок, які стосуються слова mount
card ls --output=ls.ps вивести інформацію про команду ls у вигляді карточки в Post-script форматі, яку можна роздрукувати lpr ls.ps.

Пошук окремих команд:
type mount показати першу команду mount в PATH
whereis mount показати binary, source і man сторінки для mount
which umount знайти umount в любому місці файлової системи
rpm –gal | grep umount знайти umount в любих інстальованих пакетах
rpm –g --whatprovides tar знайти пакет, який підтримує команду tar
type <command> - визначення належності команди до Linux чи shell

Виконання команд групами:

```
command-1;command-2;command-3
{command-1;command-2} > test.txt
```

Групи команд ОС Linux. ОС Linux має понад 530 команд різного призначення, які за функціональним призначенням об'єднуються у різні групи: файли і каталоги, запуск завдань і керування процесами, клавіатура, шрифти, мережі, пошта, Usenet і InterNews, X Window, робота з зображеннями, адміністративні задачі, користувачі (паролі) і групи, розроблення програмного забезпечення, службові інформаційні команди і т.п. Для спрощення пошуку команди об'єднані у більш спеціалізовані групи:

- системна інформація;
- встановлення системи;
- файли і каталоги;
- пошук файлів;
- монтування файлових систем;
- дисковий простір;
- користувачі і групи;
- встановлення/зміна повноважень на файли;
- спеціальні атрибути файлів;
- архівування і стискання файлів;
- RPM пакети (Fedora, Red Hat і т.п.);
- YUM – засіб обновлення пакетів(Fedora, RedHat і т.п.);
- DEB пакети (Debian, Ubuntu і т.п.);
- APT – засіб керування пакетами (Debian, Ubuntu и т.п.);
- Pacman – засіб керування пакетами (Arch, Frugalware і т.п.);
- перегляд вмісту файлів;
- маніпуляції з текстом;
- перетворення наборів символів і файлових форматів;
- аналіз файлових систем;
- форматування файлових систем;
- swap-простір;
- створення резервних копій (backup);
- CDROM;
- мережа (LAN і WiFi);
- мережа Microsoft Windows(SAMBA);
- міжмережевий екран IPTABLES (firewall);
- моніторинг і налагодження;
- інші корисні команди.

1.1 Команди Linux впорядковані за алфавітним порядком

(аналогічні команди вбудовані в оболонку `Bash-4.2` підкреслені)

alias - створення синонімів (alias) команд
apropos - пошук заданого слова в описаннях команд
awk (gawk) - потоковий редактор з вбудованою мовою програмування
bg - виконання завдань (jobs) у фоновому режимі
break - вихід з циклу
bind - керування розкладкою клавіатури
builtin - виконання з консолі вбудованих команд оболонки (shall)
bzip2 блоко-орієнтований файловий стискач/розтискач файлів
cal - виведення на екран календаря
caller – повертає контекст любого активного виклику підпрограми
case - виконання команд за умовою (застаріла !)
cat - об'єднати файли або стандартні входи і направити на стандартний вихід
cd - зміна робочого каталогу
cfdisk - керування таблицею розбивки жорсткого диска
chgrp - зміна власника групи
chmod - зміна бітів режиму доступу до файлу
chown - зміна власника і/або групи файлу
chroot - виконання команд і інтерактивної оболонки із із спеціального нового root каталогу
cksum - виведення контрольних CRC сум і кількості байтів
clear - очистка екрану
cmp - побайтове порівняння двох файлів
comm - порядкове порівняння двох файлів
command - виконання Linux команд, ігноруючи вбудовані команди і функції оболонки
compgen – генерація можливих співпадінь для слова відповідно до опцій
complete – описання завершення аргументів для кожного імені
compopt – модифікація опцій завершення
continue - пропуск поточної ітерації
cp - копіювання файлів і каталогів
cron - демон виконання програм за розкладом
crontab - підтримка crontab файлів для індивідуальних користувачів
csplit - розбивка файлу на секції за заданим шаблоном
cut – виведення заданих полів або записів вхідного файлу на стандартний вивід
date - виведення або зміна дати і часу
dc - потоковий калькулятор для чисел з фіксованою і плаваючою крапкою
dd - конвертування і копіювання файлів
declare - оголошення змінних і/або визначення їх атрибутів
df - використання файлового дискового простору
diff - порядкове порівняння двох файлів
diff3 - порядкове порівняння трьох файлів
dir - виведення вмісту каталогу
dircolors - символьні імена кольорів які розпізнає Tk
dirname - виділення із повного шляху тільки шляху каталогів
dirs - виведення поточного стеку каталогів
disown – робота з таблицею активних завдань (jobs)
du - довідка про використання файлового простору
echo – виведення на екран аргументів розділених пропуском з добавлення newline
egrep (grep, frep) - друк рядків файлу, які співпадають із заданим шаблоном
eject - вийняти змінний пристрій (CD-ROOM, гнучкий диск)
enable - заборона або дозвіл виконання вбудованих команд оболонки
env - виконання програми у модифікованому середовищі
ethtool - виведення і зміна встановлень ethernet карт
eval - оцінка декількох команд/скриптів. Виконання команд після виконання підставлень/розширень в оболонці
exec - виконання команд. Вихід з оболонки і перемикання на нову програму
exit - вихід з оболонки і перемикання на нову програму
expand - заміна табуляції на пропуски
export – встановлення змінних середовища. Підтримку експорту змінних
expr - оцінювання виразів
false – фальш. Повернення ненульового коду завершення
fc - оброблення списку історії команд
fdformat - низькорівневе форматування гнучких дисків
fdisk - робота з розділами жорсткого диску в Linux
fg - виконання задач у фоновому режимі

file – інформація про тип файлу
find - пошук файлів у ієрархії каталогів, які відповідають заданим критеріям
fmt - простий форматувач тексту
fold - перенесення рядків при досягненні заданої довжини
for – команда циклу
format - форматування символьних рядків у стилі sprintf
free - інформація про використання оперативної пам'яті
fsck - тестування і відновлення файлової системи
ftp - протокол передачі файлів
function - оголошення функції у сценаріях
gawk - потоковий редактор і мова програмування awk
getopts - програма видобування аргументів із списку параметрів командного рядка
grep - пошук у файлах символьних рядків , які співпадають із заданим шаблоном
groups - виведення імен груп в які входить користувач
gzip, gunzip – програми стискання і розтискання файлів
hash - запам'ятовування у змінних середовища повного шляху викликуваних команд
head - виведення на друк початкової частини файла (файлів)
help – довідкова система
history – історія списку команд введених з консолі
hostname – виведення або встановлення імені комп'ютера
id – реальні і ефективні ідентифікатори користувача і групи
if - задання умови виконання команд в сценаріях
import - захоплення вікна X сервера і збереження зображення у файлі
install - копіювання файлів і встановлення атрибутів
jobs - статус завдань у поточній сесії
join - об'єднання рядків двох файлів
kill - завершення процесу
less - поекранне/порядкове виведення вмісту файла з можливістю переміщення вперед/назад
let - виконання операцій над арифметичними виразами
ln - встановлення зв'язків між файлами
local – створення змінних. Оголошення локальних змінних у функціях
logname - виведення імені зареєстрованого користувача
logout - вихід із оболонки. Вихід із оболонки в інтерактивному режимі
look – виведення рядка з файлу за заданим символьним рядком
lpc - програма керування порядковою друкаркою
lpr - друк файлів
lprm - відміна завдань на друк
ls - виведення вмісту каталога
lsof - виведення списку відкритих файлів
make - утиліти перекомпіляції груп програм
man - довідкова система man
mapfile -> readarray
mkdir - створення нового каталогу
mkfifo - створення каналів (іменованих) типу FIFO
mkisofs - створення гібридної файлової системи ISO9660/Joliet/HFS
mknod - створення блоку або файлів спеціальних символів
more - поекранне виведення вмісту файла з можливістю порядкового переміщення вперед
mount - монтування файлової системи
mtools - утиліта роботи з MS-DOS файлами
mv - пересилання і перейменування файлів або каталогів
nc – (netcat) утиліта для встановлення мережевих з'єднань TCP, UDP
netstat - інформація про мережу, мережеві з'єднання, інтерфейсну статистику
nice - встановлення пріоритету команд або завдань
nl - нумерування рядків файлу
nohup - виконання команди нечутливої до сигналів hangups
passwd - зміна пароля користувача
paste - об'єднання рядків файлів
pathchk - перевірка допустимості імені файлу
ping - перевірка мережевого з'єднання
popd – вилучення із стеку каталогів назви поточного каталогу
pr - підготовка файлів до друку
printenv - виведення змінних середовища

printf - форматування і виведення даних. Форматоване виведення даних
ps - інформація поточні процеси
pushd – добавити ім'я каталогу у стек каталогів
pv – pipeview, команда для робот из каналами (pipes)
pwd - виведення абсолютного шляху і назви поточного каталогу
quotactl – встановлення дискових квот
ram - дисковий пристрій типу RAM
read - читання одного рядка із стандартного вводу. Читання із стандартного вводу або файлу
readlink – повний шлях до файлу
readonly - призначення іменам змінних/функціях атрибуту "тільки для читання".
return – повернення з функцій
rm – вилучення файлів або каталогів
rmdir – вилучення каталогів
rsync – локальне або віддалене копіювання (дзеркальне) файлів
screen - менеджер термінального вікна
scp - віддалене безпечне копіювання файлів
sdiff – об'єднання порядкової різниці двох файлів
sed - потоковий редактор
seq - виведення числових послідовностей
set - встановлення опцій для імен і значень змінних
sftp - програма безпечного пересилання файлів
shift - зсув позиційних параметрів
shopt – перемикання значень змінних які встановлюють властивості оболонки Bash
shutdown - завершення або перевантаження Linux
sleep - затримка на заданий проміжок часу
sort - сортування текстових файлів
source - виконання команд файлу з '.'
split - розбиття файлу на частини фіксованих розмірів
ssh - клієнт безпечної оболонки (програма віддаленої реєстрації в системі)
stat – детальна інформація про файли і файлову систему
strace - трасування системних викликів і сигналів
su - виконання програм з правами інших користувачів
sum - друк контрольних сум файлів і підрахунок кількості блоків у файлах
suspend – затримка виконання оболонки до отримання сигналу SIGCONT
symlink - створення нового символьного імені файлу (нового шляху, який містить старий шлях)
sync - скидання буфера файлової системи (суперблоку) на диск
tail - виведення останньої частини файлу
tar - утиліта створення архіву типу tar
tee – перенаправлення виводу на стандартний вихід і файл (або декілька файлів)
test - оцінка умовних виразів і повернення результати 0 або 1
time - визначення часу виконання команд
times - час системи і користувача. Виведення статистики часу виконання оболонки
touch - - зміна часових атрибутів файлу, створення порожнього файлу
top - виведення діючих процесів Linux
traceroute - трасування маршрутів пакетів до мережевого хоста
trap – виконання команд при обробленні і захоплення сигналів
tr - трансляція або вилучення символів із вхідного потоку
true - логічне значення істина
tsort - топологічне сортування
tty - виведення імені терміналу на stdin
type – описання команди. Визначення типу імен, при використанні їх як команд
typeset - оголошення змінних
ulimit - встановлення і отримання обмежень для користувача. Встановлення контролю над ресурсами оболонки
umask - встановлення і отримання обмежень для користувача. Задання режиму створення файлів користувачем
umount - розмонтування пристрою
unalias – вилучення аласів. Вилучення імені із списку аліасів
uname - виведення інформації про систему
unexpand - перетворення пропусків у табуляцію
uniq - виведення або пропущення рядків які повторюються у файлах
units – перетворення одиниць вимірювань з однієї шкали в іншу

unset - вилучення змінних або функцій за заданим іменем
unshar - розпакування архівів shell сценаріїв
until – виконання команд (до помилки)
useradd - створення нового облікового запису користувача
usermod - корегування облікового запису користувача
users - виведення імен користувачів, зареєстрованих у системі
uuencode - кодування бінарного файлу
uudecode – декодування файлу створеного uuencode
vdir - виведення вмісту каталогу ('ls -l –b')
vi – текстовий редактор
wait – очікувати на завершення заданого процесу і повернути його код завершення
watch – виконання або виведення результатів роботи програми періодично
wc - виведення кількості байт, слів і рядків
whereis - пошук всіх появлень команди у файлах
which - пошук програми у шляху користувача
while - виконання сценарію за умовою циклу
who - виведення інформації про зареєстрованих користувачів
whoami - виведення id і імені поточного користувача
wget - неінтерактивне завантаження веб-сторінок або файлів через HTTP, HTTPS, FTP

xargs - побудова і виконання командного рядку із стандартного вводу
yes - виведення у нескінченому циклі символьного рядка (до зняття CTRL+C)
.(крапка) - виконання команд з файлу
- коментар

2 Системна інформація

arch або **uname -m** — відобразити архітектуру комп'ютера
uname -r — відобразити версію ядра ОС
dmidecode -q — показати апаратні системні компоненти — (SMBIOS / DMI)
hdparm -i /dev/hda — вивести характеристики жорсткого диска
hdparm -tT /dev/sda — протестувати продуктивність читання даних з жорсткого диска
cat /etc/os-release – інформація про версію ОС
cat /etc/SuSE-release - інформація про версію ОС Suse
cat /proc/cpuinfo — відобразити інформацію про процесор
cat /proc/interrupts — показати переривання
cat /proc/meminfo — перевірити використання пам'яті
cat /proc/swaps — показати файл(и) підкачування
cat /proc/version — вивести версію ядра
cat /proc/net/dev — показати мережеві інтерфейси і їх статистику
cat /proc/mounts — відобразити змонтовані файлові системи
clear - очищення екрану термінала
lspci -tv — показати як дерево PCI пристрої
lsusb -tv — показати як дерева USB пристрої
date — вивести системну дату
cal -3 – календар попереднього, поточного і наступного місяця
uptime - поточний час та робота системи без перевантаження і виключення
cal 2020 — вивести таблицю-календар 2020-го року
date 041217002007.00* — встановити системну дату і час ММДДГГххРРРР.СС (МісяцьДеньГодиниХвилиниРік.Секунди)
clock -w — зберегти системний час у BIOS
зупинка системи:
shutdown -h now або init 0 або telinit 0 — зупинити систему
shutdown -h now - вихід з Linux
shutdown -h hours:minutes & — запланувати запинку системи у вказаний час
shutdown -c — відмінити заплановану за розкладом зупинку системи
shutdown -r now або **reboot** — перевантажити систему
logout — вийти із системи
poweroff - вихід з Linux
reboot - перевантаження системи
last reboot - статистика останніх перевантажень
whois linux.org - інформація про домен linux.org
history – історія команд
history | tail -10 – показати останні 10 введені команди

!! - виконати останню команду
!n – виконати команду з номером n
history !номер – виконати команду за номером
fc –s номер – виконати команду за номером
exit - завершити сеанс поточного користувача
passwd - змінити пароль поточного користувача
winecfg - налаштування Wine (не емулятор WinAPI)
finger, users, who, w – інформація про користувачів системи
lsmod – список модулів завантажених у ядро

3 Файли і каталоги

cd /home — перейти в каталог '/home'
cd $HOME - перейти в home каталог
cd ~ - перейти в home каталог
cd .. — перейти в каталог рівнем вище
cd ../.. — перейти в каталог двома рівнями вище
cd $OLDPWD перейти в попередній робочий каталог
cd — перейти в домашній каталог
cd ~user — перейти в домашній каталог користувача user
cd - — перейти в каталог, в якому знаходилися до переходу у поточний каталог
cd /dev; ls –al sda* ttyS* - виведення списку файлів пристроїв
cd /usr/bin - перейти в каталог usr/bin з root каталогу
pwd — показати поточний каталог
ls / - список каталогів файлової системи
ls – виведення впорядкованого списку каталогів і файлів
ls -F — відобразити вміст поточного каталогу з добавленням до імен символів, які характеризують тип
ls -l - інформація про права доступу і власників. Перший символ задає тип файлу (d-каталог, l-символьне посилання, s-сокет, b-блоковий пристрій, c-символьний пристрій, p-іменований канал), решта символів задають права доступу власника, групи і інших користувачів rwx rwx rwx, де r-читання, запис, x-виконання.
ls -a — показати скриті файли і каталоги в поточному каталозі
ls –las - виведення відсортованого за іменами списку усіх каталогів і файлів
ls –laX - виведення відсортованого за розширеннями списку усіх каталогів і файлів
ls *[0-9]* — показати файли і каталоги, які містять у імені цифри
tree или **lstree** — показати дерево файлів і каталогів, починаючи з кореня (/)
mkdir dir1 — створити каталог з іменем 'dir1'
mkdir dir1 dir2 — створити два каталоги одночасно
mkdir -p /tmp/dir1/dir2 — створити дерево каталогів
rm -f file1 — вилучити файл з іменем 'file1'
rmdir dir1 — вилучити каталог з іменем 'dir1'
rm -rf dir1 — вилучити каталог з іменем 'dir1' і рекурсивно увесь його вміст
rm -rf dir1 dir2 — вилучити два каталоги і рекурсивно їх вміст
mv dir1 new_dir — перейменувати чи перемістити файл або каталог
cp file1 file2 — скопіювати файл file1 у файл file2
cp dir/* . — копіювати всі файли каталогу dir в поточний каталог
cp -a /tmp/dir1 . — скопіювати каталог dir1 із усім вмістом у поточний каталог
cp -a dir1 dir2 — копіювати каталог dir1 у каталог dir2
ln -s file1 lnk1* — створити «м'яке» (символьне) посилання на файл або каталог
ln file1 lnk1 — створити «жорстке» (фізичне) посилання на файл або каталог
touch -t 0712250000 fileditest — модифікувати дату і час створення файлу, а при його відсутності, створити файл з вказаними датою і часом (YYMMDDhhmm)
touch file – створити порожній файл
touch hello.$$ - створити порожній файл з унікальним розширенням
touch $$hello – створити порожній файл з унікальним іменем
head /var/log/messages - вивести початок файлу
tail /var/log/messages - вивести кінець файлу (для великих файлів)
more file – по екранне виведення файлу з рухом вперед
less file – по екранне виведення файлу з рухом назад
echo "Останній рядок" | sudo tee -a /home/text - добавлення текстового рядка "Останній рядок" у кінець файлу /home/text

cp /home/user1/my.txt /home/new.txt - копіювати файл /home/user1/my.txt у файл home/new.txt
ln /user/file1 /user/file2 - створити жорсткий зв'язок між файлами (hard link) (аналог до cp –l /user/file1 /user/file2)
ln –s /user/file1 /user/file2 - створити м'який зв'язок між файлами (soft link) (аналог до cp –s /user/file1 /user/file2)
mkdir /home/user1/newdir – створення нового каталогу /home/user1/newdir
rmdir /home/dir – вилучити каталог dir
rm -rf /home/user1/dir - вилучити каталог dir з вкладеними каталогами
cp -la /dir1 /dir2 - копіювати каталог dir1 в каталог dir2
mv /dir1 /dir2 - перейменувати каталог dir1 в каталог dir2
echo "Останній рядок" | sudo tee –a /home/file – добавити повідомлення в останній рядок файлу file
du -sh /home/Documents – визначення розміру каталогу
du -sh /home/file – визначення розміру файлу
touch /home/newfile – створення порожнього файлу /home/newfile
> /home/newfile – створення порожнього файлу Ctrl+D – кінець введення)
cat > newfile - створення порожнього файлу (Ctrl+D – кінець введення)
mv /home/file1 /home/file2 – перейменування файлів
rm /home/file – вилучення файлу
cp /home/file1 /home/file2 – копіювання файлу
locate file – пошук усіх файлів з іменем file
dd if=/dev/zero of=/tmp/mynullfile count=1 - створення null-файлу (/dev/zero є спеціальний файл, який генерує null символи, count – лічильник блоків, за замовчуванням блок має 512 байт)
od -vt x1 /tmp/mynullfile - перегляд 8-го дампу файла
dd if=/dev/hda of=mymbrfile bs=512 count=1 – копіювання MBR із завантажувального сектора IDE жорсткого диска
dd if=/dev/cdrom of=whatever.iso - копіювання ISO образа із CD або DVD
vim, kwrite – редагування файлів з використання редакторів
sort – сортування рядків файлу
nl - нумерація рядків файлу
grep – друк рядків файлу, які співпадають з шаблоном
cmp file1, file2 – порівняння файлів
diff file.old, file.new – порівняння файлів і виведення їх розходжень
diff file.old, file.new > dif.txt – виведення розходжень файлів у файл
find file – пошук файлу в ієрархії каталогів
find / -name e100 -print 2> /dev/null – пошук файлу e100, починаючи з root каталогу і направлення повідомлень про помилки в нульовий пристрій
cat file – виведення вмісту файлу
stat – виведення параметрів файлу, прав і часу доступу до файлу
file – виведення типу файлу
mkfifo – створення іменованого каналу (черга типу FIFO)
echo "Команда для роботи з каналом" | pv –qL 5 посимвольне виведення стрічки із швидкістю 5 байт/сек
cat file.txt | pv –s `du –sb file.txt | cut –f1` виведення файлу на екран з візуалізацією прогресу виконання в %.
wget http://itshaman.ru/images/logo_white.png - скачати файл logo_white.png у поточну папку
wget --convert-links -r http://www.linux.org/ - копіювати сайт повністю (на 5 рівнів в глибину) і конвертувати посилання для автономної роботи
md5sum openSUSE-10.3-RC1-KDE-i386.iso - перевірка контрольної суми файлу
sha1sum openSUSE-10.3-RC1-KDE-i386.iso - перевірка контрольної суми файлу
zip, bzip2 – створення архівів файлів
tar –cf arx.tar file1, file2, file3- створення tar-архіву заданих файлів
tar –cf arx.tar *.* - створення tar-архіву усіх файлів поточного каталогу
tar –cvf arx.tar subdir - створення tar-архіву підкаталогу
gzip file - створення архіву файла

4 Пошук файлів

find / -name file1 — знайти файли і каталоги з іменем file1. Пошук почати з кореня (/)

find / -user user1 – знайти файл і каталог, які належать користувачу user1. Пошук почати з кореня (/)
find /home/user1 -name "*.bin" – знайти всі файли і каталоги, імена яких закінчуються на '. bin'. Пошук почати з '/ home/user1'*
find /usr/bin -type f -atime +100 – знайти всі файли в '/usr/bin', час останнього звернення до яких не більше 100 днів
find /usr/bin -type f -mtime -10 – знайти всі файли в '/usr/bin', які створені або змінені за останні 10 днів
find / -name *.rpm -exec chmod 755 '{}' \; – знайти всі файли і каталоги, імена яких закінчуються на '.rpm', та змінити права доступу до них
find / -xdev -name "*.rpm" – знайти всі файли і каталоги, імена яких закінчуються на '.rpm', ігноруючи знімні носії, такі як cdrom, floppy і т.п.
locate "*.ps" – знайти всі файли, які містять в імені '.ps'. Попередньо рекомендується виконати команду 'updatedb'
whereis halt – показати розміщення бінарних файлів, сирцевих кодів і керівництв, які відносяться до файлу 'halt'
which halt – відобразити повний шлях до файлу 'halt'

5 Перегляд і редагування файлів

grep -HR OLDTEXT ./ | awk '{print $1}' | sed 's/:.*$//' | grep -v '~' | sort | uniq | xargs perl -i -pe "s/OLD_TEXT/NEW_TEXT/g;" – Пошук і заміна тексту OLDTEXT на NEW_TEXT у багатьох файлах одночасно с рекурсивним обходом каталогів
cat file_originale | [operation: sed, grep, awk, grep и т.п.] > result.txt – загальний синтаксис виконання дій по обробленні вмісту файла і виведення результату в новий файл
cat file_originale | [operazione: sed, grep, awk, grepи т.п.] >> result.txt – загальний синтаксис виконання дій по обробленні вмісту файлу і виведенню результату в існуючий файл. Якщо файл не існує, він буде створений
grep Aug /var/log/messages – із файлу '/var/log/messages' відобразити і вивести на стандартний пристрій виведення стрічки, яка містить «Aug»
grep ^Aug /var/log/messages – із файлу '/var/log/messages' вибрати і вивести на стандартний пристрій виведення стрічки, яка починається з «Aug»
grep [0-9] /var/log/messages – із файлу '/var/log/messages' вибрати і вивести на стандартний пристрій виведення стрічки, яка містить цифри
grep Aug -R /var/log/* – вибрати і вивести на стандартний пристрій виведення стрічки, яка містить «Augr», у всіх файлах, які знаходяться в каталозі /var/log і нище
sed 's/stringa1/stringa2/g' example.txt – у файлі example.txt замінити «string1» на «string2», результат вивести на стандартний пристрій виведення
sed '/^$/d' example.txt – вилучити порожні рядки із файлу example.txt
sed '/ *#/d; /^$/d' example.txt – вилучити порожні рядки і коментарі з файлу example.txt
echo 'esempio' | tr '[:lower:]' '[:upper:]' – перетворити символи з нижнього регістра у верхній
sed -e '1d' result.txt – вилучити перший рядок з файлу example.txt
sed -n '/string1/p' – відобразити тільки рядки, які містять «string1»
sed -e 's/ *$//' example.txt – вилучити порожні символи у кінці кожного рядка
sed -e 's/string1//g' example.txt – вилучити стрічку «string1» з тексту не змінюючи всього іншого
sed -n '1,8p;5q' example.txt – взяти з файлу з першого по восьмий рядок і з них вивести перші п'ять рядків
sed -n '5p;5q' example.txt – вивести п'ятий рядок
sed -e 's/0*/0/g' example.txt – замінити послідовність з любої кількості нулів одним нулем
cat -n file1 – перенумерувати рядки при виведенні вмісту файлу
cat example.txt | awk 'NR%2==1' – при виведенні вмісту файлу, не виводити парні рядки
echo a b c | awk '{print $1}' – вивести перший стовпець. Розділення стовпців, за замовчуванням, за забілом/забілами або символом/символами табуляції
echo a b c | awk '{print $1,$3}' – вивести перший і третій стовпці Розділення стовпців, за замовчуванням, за забілом/забілами або символом/символами табуляції

paste file1 file2 — об'єднати вміст file1 і file2 як таблицю: рядок 1 з file1 = рядок 1 стовпець 1-n, рядок 1 з file2 = рядок 1 стовпець n+1-m
paste -d '+' file1 file2 — об'єднати вміст file1 і file2 як таблицю з розділювачем «+»
sort file1 file2 — відсортувати вміст двох файлів
sort file1 file2 | uniq — відсортувати вміст двох файлів, не відображуючи повторень
sort file1 file2 | uniq -u — відсортувати вміст двох файлів, відображуючи тільки унікальні рядки (рядки, які повторюються в обох файлах, не виводяться на стандартний пристрій виведення)
sort file1 file2 | uniq -d — відсортувати вміст двох файлів, відображуючи тільки рядки, які повторюються
comm -1 file1 file2 — порівняти вміст двох файлів, не відображуючи рядки, які належать файлу 'file1'
comm -2 file1 file2 — порівняти вміст двох файлів, не відображуючи рядки, які належать файлу 'file2'
comm -3 file1 file2 — порівняти вміст двох файлів, вилучаючи рядки, які зустрічаються в обох файлах

6 Архівування і стискання файлів

bzip2 myfile — стиснути файл з найбільш можливим ступенем стиску
bunzip2 file1.bz2 — розтиснути файл 'file1.bz2'
gzip file1 или **bzip2 file1** — стиснути файл 'file1'
gzip -9 file1 — максимально стиснути файл file1
gunzip file1.gz — розтиснути файл 'file1.gz'
lzop –v myfile — стиснути файл з найбільшою швидкістю
lzop –d myfile.lzo — розстиснути файл
rar a file1.rar test_file — створити rar-архів 'file1.rar' і включити в нього файл test_file
rar a file1.rar file1 file2 dir1 — створити rar-архів 'file1.rar' і включити в нього file1, file2 і dir1
rar x file1.rar — розпакувати rar-архів
unrar x file1.rar — розпакувати rar-архів
tar -cvf archive.tar file1 — створити tar-архів archive.tar, який містить файл file1
tar -cvf archive.tar file1 file2 dir1 — створити tar-архів archive.tar, який містить файли file1, file2 і каталог dir1
tar -tf archive.tar — показати вміст архіву
tar -xvf archive.tar — розпакувати архів
tar -xvf archive.tar -C /tmp — розпакувати архів в /tmp
tar -cvfj archive.tar.bz2 dir1 — створити архів і стиснути його за допомогою bzip2
tar -xvfj archive.tar.bz2 — розтиснути архів і розпакувати його
tar -cvfz archive.tar.gz dir1 — створити архів і стиснути його за допомогою gzip
tar -xvfz archive.tar.gz — розтиснути архів і розпакувати його
zip file1.zip file1 — створити стиснутий zip-архів
zip -r file1.zip file1 file2 dir1 — створити стиснутий zip-архів і помістити в нього файли і/або каталоги
unzip file1.zip — розтиснути і розпакувати zip-архів RPM пакети (Fedora, Red Hat і т.і):

7 Дисковий простір

df -h — відобразити інформацію про змонтовані розділи з відображенням загального, доступного та використовуваного простору
ls -lSr | more — вивести список файлів і каталогів рекурсивно з сортуванням за зростанням розміру і дозволяє здійснити посторінковий перегляд
du -sh dir1 — підрахувати і вивести розмір, який займає каталог 'dir1'
du -sk * | sort -rn — відобразити розмір і імена файлів і каталогів, з сортуванням за разміром
rpm -q -a --qf '%10{SIZE}t%{NAME}n' | sort -k1,1n — показати розмір використовуваного дискового простору, який займають файли rpm-пакету, з сортуванням за розміром (fedora, redhat і т.п.)

dpkg-query -W -f='${Installed-Size;10}t${Package}n' | sort -k1,1n — показати розмір використовуваного дискового простору, який займають файли deb-пакету, з сортуванням за розміром (ubuntu, debian і т.п.)
Користувачі і групи:
groupadd group_name — створити нову групу з іменем group_name
groupdel group_name — вилучити групу group_name
groupmod -n new_group_name old_group_name — перейменувати групу old_group_name в new_group_name
useradd -c "Nome Cognome" -g admin -d /home/user1 -s /bin/bash user1 — створити користувача user1, назначити йому як домашній каталог /home/user1, а як оболонки (shell) /bin/bash, включити його в групу admin і добавити коментарій Nome Cognome
useradd user1 — створити користувача user1
userdel -r user1 — вилучити користувача user1 і його домашній каталог
usermod -c "User FTP" -g system -d /ftp/user1 -s /bin/nologin user1 — змінити атрибути користувача
passwd — змінити пароль
passwd user1 — змінити пароль користувача user1 (тільки root)
chage -E 2005-12-31 user1 — встановити дату закінчення дії облікового запису користувача user1
pwck — перевірити коректність системних файлів облікових записів. Перевіряються файли /etc/passwd і /etc/shadow
grpck — перевірити коректність системних файлів облікових записів. Перевіряється файл/etc/group
newgrp [-] group_name — змінити первинну групу поточного користувача. Якщо вказати «-», то ситуація буде ідентичною до тієї, в якій користувач вийшов із системи і знову зайшов. Якщо не вказати групу, первинна група буде назначена з /etc/passwd
Встановлення/зміна повноважень на файли:
ls -lh — перегляд повноважень на файли і каталоги в поточному каталозі
ls /tmp | pr -T5 -W$COLUMNS — вивести вміст каталогу /tmp і розділити виведення на п'ять стовпців
chmod ugo+rwx directory1 — добавити повноваження на каталог directory1 ugo(User Group Other)+rwx(Read Write eXecute) — усім повні права. Аналогічний результат можна отримати командою chmod 777 directory1
chmod go-rwx directory1 — відібрати у групи і всіх інших всі повноваження на каталог directory1.
chown user1 file1 — призначити власником файлу file1 користувача user1
chown -R user1 directory1 — призначити рекурсивно власником каталогу directory1 користувача user1
chgrp group1 file1 — змінити групу-власника файлу file1 на group1
chown user1:group1 file1 — змінити власника і групу власника файлу file1
find / -perm -u+s — знайти, починаючи від кореня, всі файли з встановленим SUID
chmod u+s /bin/binary_file — встановити SUID-біт файлу /bin/binary_file. Це дає можливість любому користувачу запускати на виконання файл з повноваженнями власника файлу
chmod u-s /bin/binary_file — зняти SUID-біт з файлу /bin/binary_file
chmod g+s /home/public — встановити SGID-біт каталогу /home/public
chmod g-s /home/public — зняти SGID-біт з каталогу /home/public
chmod o+t /home/public — призначити STIKY-біт каталогу /home/public. Дозволяє вилучати файли тільки власникам
chmod o-t /home/public — зняти STIKY-біт з каталогу /home/public
Спеціальні атрибути файлів:
chattr +a file1 — дозволити відкривати файл на запис тільки в режимі добавлення
chattr +c file1 — дозволити ядру автоматично стискати/розтискати вміст файлу
chattr +d file1 — вказати утиліті dump ігнорувати даний файл під час виконання backup'а
chattr +i file1 — зробити файл недоступним для любих змін: редагування, вилучення, переміщенння, створення посилань на нього
chattr +s file1 — зробити вилучення файлу безпечним, тобто встановлений атрибут s вказує на те, що при вилученні файлу, місце, яке займає файл на диску заповнюється нулями, що запобігає можливості відновлення даних
chattr +S file1 — вказати, що, при збереженні змін, буде виконана синхронізація, як при виконанні команди sync
chattr +u file1 — вказати, що при вилученні файлу його вміст буде збережений и при необхідності користувач зможе його відновити

lsattr – показати атрибути файлів

8 Встановлення програмних пакетів в SUSE

Для встановлення програмних пакетів в Linux використовують наступні засоби:

• rpm – створення і керування програмними пакетами;

• zypper – інструмент командного рядка для підключення до репозиторіїв, оновлення пакетів з автоматичним розв'язуванням їх залежностей від інших програм і бібліотек;

• YAST – тексто-орієнтований або графічний інтерфейс для завантаження і встановлення програмних пакетів з online репозиторіїв.

Синтаксис команди rpm:

rpm <опції>

-i – інсталювати пакет;
-e – вилучити пакет;
-U – оновити пакет;
-F – оновити вже встановлений пакет;
-a – перевірити всі пакети.

rpm -ivh package.rpm – встановити пакет з виведенням повідомлень і індикатора процесу виконання
rpm -ivh --nodeps package.rpm – встановити пакет з виведенням повідомлень індикатора процесу виконання без контролю залежностей
rpm -U package.rpm – обновити пакет без зміни конфігураційних файлів, у випадку відсутності пакету
rpm -F package.rpm – обновити пакет тільки якщо він встановлений
rpm -e package_name.rpm – вилучити пакет
rpm -qa – відобразити список всіх пакетів, встановлених в системі
rpm -qa | grep httpd – знайти пакет, який містить в своєму імені «httpd», серед всіх пакетів, встановлених у системі
rpm -qi package_name – вивести інформацію про конкретний пакет
rpm -qg "System Environment/Daemons" – відобразити пакети, які входять в групу пакетів
rpm -ql package_name – вивести список файлів, які входять в пакет
rpm -qc package_name – вивести список конфігураційних файлів, які входять в пакет
rpm -q package_name --whatrequires – вивести список пакетів, необхідних для встановлення конкретного пакету за залежностями
rpm -q package_name --whatprovides – показати можливості rpm пакету
rpm -q package_name --scripts – відобразити сценарії, які запускаються при встановленні/вилученні пакету
rpm -q package_name --changelog – вивести історію ревізії пакету
rpm -qf /etc/httpd/conf/httpd.conf – перевірити якому пакету належить вказаний файл. Вказувати потрібно повний шлях і ім'я файлу
rpm -qp package.rpm -l – відобразити список файлів, які входять в пакет, але ще не встановлені у систему
rpm --import /media/cdrom/RPM-GPG-KEY – імпортувати публічний ключ цифрового підпису
rpm --checksig package.rpm – перевірити підпис пакета
rpm -qa gpg-pubkey – перевірити цілісність вмісту встановленого пакету
rpm -V package_name – перевірити розмір, повноваження, тип, власника, групу, MD5-суму і дату останньої зміни пакету
rpm -Va – перевірити вміст всіх пакетів встановлених у систему. Виконувати обережно!
rpm -Vp package.rpm – перевірити пакет, який ще не встановлений у систему
rpm2cpio package.rpm | cpio --extract --make-directories *bin* – видобути з пакету файли, які містять у своєму імені bin
rpm -ivh /usr/src/redhat/RPMS/`arch`/package.rpm – встановити пакет, зібраний із сирцевих кодів
rpmbuild --rebuild package_name.src.rpm – зібрати пакет із сирцевих кодів

Синтаксис команди `zypper`:

zypper [опції] команди [опції] аргументи

Групи команд:
- керування репозиторієм:
  - `repos` – вивести всі визначені репозиторії;
  - `addrepo` – додати новий репозиторій.
  - `removerepo` – вилучити вказаний репозиторій.
  - `renamerepo` – перейменувати вказаний репозиторій
  - `modifyrepo` – модифікувати вказаний репозиторій
  - `refresh` – оновити всі репозиторії;
  - `clean` – очистка локальних кешів.
- керування сервісами;
- керування програмами:
  - `install -` інсталювати пакети;
  - `remove` – вилучити пакети;
  - `verify` – перевірити цілісність залежностей пакету.
- керування оновленням:
  - `upgrate -` оновити встановлені пакети;
  - `upgrate -` встановити потрібні латки;
  - `patch-check` – перевірити латки.

9 Встановлення програмних пакетів в Fedora, RedHat (YUM)

yum install package_name — завантажити і встановити пакет
yum update — обновити всі пакети, встановлені в систему
yum update package_name — обновити пакет
yum remove package_name — вилучити пакет
yum list — вивести список всіх пакетів, встановлених у системі
yum search package_name — знайти пакет у репозиторіях
yum clean packages — очистити rpm-кеш, вилучивши завантажені пакети
yum clean headers — вилучити всі заголовки файлів, які система використовує для разв'язування залежностей
yum clean all — очистити rpm-кеш, вилучивши закачані пакети і заголовки

10 Встановлення програмних пакетів в Debian, Ubuntu (DEB)

dpkg -i package.deb — встановити / обновити пакет
dpkg -r package_name — вилучити пакет з системи
dpkg -l — показати всі пакети, встановлені в систему
dpkg -l | grep httpd — серед всіх пакетів, встановлених в системі, знайти пакет, який містить в своєму імені «httpd»
dpkg -s package_name — відобразити інформацію про конкретний пакет
dpkg -L package_name — вивести список файлів, які входять в пакет, встановлений у систему
dpkg --contents package.deb — відобразити список файлів, які входять у пакет, який ще не встановлений у систему
dpkg -S /bin/ping — знайти пакет, в який входить вказаний файл
APT — засіб керування пакетами (Debian, Ubuntu і т.п.):
apt-get update — отримати обновлені списки пакетів
apt-get upgrade — обновити пакети, встановлені в систему
apt-get install package_name — встановити / обновити пакет
apt-cdrom install package_name — встановити / обновити пакет з cdrom'а
apt-get remove package_name — вилучити пакет, встановлений в систему із збереженням файлів конфігурації
apt-get purge package_name — вилучити пакет, встановлений в систему з вилученням файлів конфігурації

apt-get check — перевірити цілісність залежностей
apt-get clean — вилучити завантажені архівні файли пакетів
apt-get autoclean — вилучити старі завантажені архівні файли пакетів
Pacman — засіб керування пакетами (Arch, Frugalware і т.п.)
pacman -S name — встановити пакет «name» із залежностями
pacman -R name — вилучити пакет «name» і всі його файли
Перегляд вмісту файлів:
cat file1 — вивести вміст файлу file1 на стандартний пристрій виведення
tac file1 — вивести вміст файлу file1 на стандартний пристрій виведення у зворотному порядку (з кінця)
more file1 — посторінкове виведення вмісту файлу file1 на стандартний пристрій виведення
less file1 — посторінкове виведення вмісту файлу file1 на стандартний пристрій виведення, але з можливістю гортання в обидві сторони (вверх-вниз), пошук за змістом і т.п.
head -2 file1 — вивести перші два рядки файлу file1 на стандартний пристрій виведення. За замовчуванням виводиться десять рядків
tail -2 file1 — вивести останні два рядки файлу file1 на стандартний пристрій виведення. За замовчуванням виводиться десять рядків
tail -f /var/log/messages — вивести вміст файлу /var/log/messages на стандартний пристрій виведення по мірі появлення в ньому тексту

11 Перетворення наборів символів і файлових форматів

dos2unix filedos.txt fileunix.txt — конвертувати файл текстового формату з MSDOS в UNIX (різниця в символах повернення каретки)
unix2dos fileunix.txt filedos.txt — конвертувати файл текстового формату з UNIX в MSDOS (різниця в символах повернення каретки)
recode ..HTML < page.txt > page.html — конвертувати вміст текстового файлу page.txt в html-файл page.html
recode -l | more — вивести список доступних форматів

12 Аналіз файлових систем

badblocks -v /dev/hda1 — перевірити розділ hda1 на наявність bad-блоків
fsck /dev/hda1 — перевірити/відновити цілісність linux-файлової системи розділу hda1
fsck.ext2 /dev/hda1 або **e2fsck /dev/hda1** — перевірити/відновити цілісність файлової системи ext2 розділу hda1
e2fsck -j /dev/hda1 — перевірити/відновити цілісність файлової системи ext3 розділу hda1 з вказівкою, що журнал розміщений там же
fsck.ext3 /dev/hda1 — перевірити/відновити цілісність файлової системи ext3 розділу hda1
fsck.vfat /dev/hda1 або **fsck.msdos /dev/hda1** або **dosfsck /dev/hda1** — перевірити/відновити цілісність файлової системи fat розділу hda11

13 Створення файлових систем

mkfs /dev/hda1 — створити linux-файлову систему на розділі hda1
mke2fs /dev/hda1 — створити файлову систему ext2 на розділі hda1
mke2fs -j /dev/hda1 — створити журнальну файлову систему ext3 на розділі hda1
mkfs -t vfat 32 -F /dev/hda1 — створити файлову систему FAT32 на розділі hda1
fdformat -n /dev/fd0 — форматування гнучкого диску без перевірки
mkswap /dev/hda3 — створення swap-простору на розділі hda3
swapon /dev/hda3 — активувати swap-простір, розміщений на розділі hda3
swapon /dev/hda2 /dev/hdb3 — активувати swap-простір, розміщений на розділах hda2 і hdb3

14 Монтування файлових систем

mount /dev/hda2 /mnt/hda2 – змонтувати розділ 'hda2' у точку монтування '/mnt/hda2'. Необхідно переконатися в наявності у каталогу точки монтування '/mnt/hda2'
umount /dev/hda2 – розмонтувати розділ 'hda2'. Перед виконанням необхідно покинути '/mnt/hda2'
fuser -km /mnt/hda2 – примусово розмонтувати розділу. Застосовується у випадку, коли розділ зайнятий яким-небудь користувачем
umount -n /mnt/hda2 – розмонтувати без занесення інформації в /etc/mtab. Корисно коли файл має атрибути «тільки читання» або недостатньо місця на диску
mount /dev/fd0 /mnt/floppy – змонтувати гнучкий диск
mount /dev/cdrom /mnt/cdrom – змонтувати CD або DVD
mount /dev/hdc /mnt/cdrecorder – монтувати CD-R/CD-RW або DVD-R/DVD-RW(+-)
mount -o loop file.iso /mnt/cdrom – змонтувати ISO-образ
mount -t vfat /dev/hda5 /mnt/hda5 – змонтувати файлову систему Windows FAT32
mount -t smbfs -o username=user,password=pass //winclient/share /mnt/share – змонтувати мережеву файлову систему Windows (SMB/CIFS)
mount -o bind /home/user/prg /var/ftp/user – змонтувати каталог в каталог (binding). Доступна з версії ядра 2.4.0. Корисна, наприклад, для надання вмісту каталогу користувача через ftp при роботі ftp-сервера в «пісочниці» (chroot), коли "симлінки" зробити неможливо. Виконання даної команди зробить копію вмісту /home/user/prg в /var/ftp/user

15 Створення резервних копій (backup)

dump -0aj -f /tmp/home0.bak /home – створити повну резервну копію каталогу /home в файл /tmp/home0.bak
dump -1aj -f /tmp/home0.bak /home – створити інкрементну резервну копію каталогу /home в файл /tmp/home0.bak
restore -if /tmp/home0.bak – відновити з резервної копії /tmp/home0.bak
rsync -rogpav --delete /home /tmp – синхронізувати /tmp з /home
rsync -rogpav -e ssh --delete /home ip_address:/tmp – синхронізувати через SSH-тунель
rsync -az -e ssh --delete ip_addr:/home/public /home/local – синхронізувати локальний каталог з віддаленого каталогу через ssh-тунель із стисканням
rsync -az -e ssh --delete /home/local ip_addr:/home/public – синхронізувати віддалений каталог з локальним каталогом через ssh-тунель із стисканням
dd bs=1M if=/dev/hda | gzip | ssh user@ip_addr 'dd of=hda.gz' – зробити «копію злiпок» локального диску в файл на віддаленому комп'ютері через ssh-тунель
tar -Puf backup.tar /home/user – створити інкрементну резервну копію каталогу '/home/user' у файл backup.tar із збереженням повноважень
(cd /tmp/local/ && tar c .) | ssh -C user@ip_addr 'cd /home/share/ && tar x -p' – копіювання вмісту /tmp/local на віддалений комп'ютер через ssh-тунель в /home/share/
(tar c /home) | ssh -C user@ip_addr 'cd /home/backup-home && tar x -p' – копіювання вмісту /home на віддалений комп'ютер через ssh-тунель в /home/backup-home
tar cf - . | (cd /tmp/backup ; tar xf -) – копіювання одного каталогу в інший із збереженням повноважень і посилань
find /home/user1 -name '*.txt' | xargs cp -av --target-directory=/home/backup/ --parents – пошук в /home/user1 всіх файлів, імена яких закінчуються на '.txt', і копіювання їх в другий каталог
find /var/log -name '*.log' | tar cv --files-from=- | bzip2 > log.tar.bz2 – пошук в /var/log всіх файлів, імена яких закінчуються на '.log', і створення bzip-архіву з них
dd if=/dev/hda of=/dev/fd0 bs=512 count=1 – створити копію MBR (Master Boot Record) з /dev/hda на гнучкий диск
dd if=/dev/fd0 of=/dev/hda bs=512 count=1 – відновити MBR з гнучкого диску на /dev/hda

16 Робота з CDROM

cdrecord -v gracetime=2 dev=/dev/cdrom -eject blank=fast -force – очистити повторно записуваний (rewritable) cdrom

mkisofs /dev/cdrom > cd.iso — створити iso образ cdrom на диску
mkisofs /dev/cdrom | gzip > cd_iso.gz — створити стиснутий iso образ cdrom на диску
mkisofs -J -allow-leading-dots -R -V "Label CD" -iso-level 4 -o ./cd.iso data_cd — створити iso образ з каталогу
cdrecord -v dev=/dev/cdrom cd.iso — записати ISO образ на cdrom
gzip -dc cd_iso.gz | cdrecord dev=/dev/cdrom — записати стиснутий ISO образ на cdrom
mount -o loop cd.iso /mnt/iso — змонтувати ISO образ
cd-paranoia -B — переписати аудіо записи з CD у wav файли
cd-paranoia -- "-3" — переписати перші три аудіо записи з CD у wav файли
cdrecord --scanbus — сканувати шину для ідентифікації scsi каналу

17 Використання ресурсів і пристроїв

dmesg – показати log-файл завантаження системи і знаходження нових пристроїв
free – інформація про використання оперативної пам’яті
top – динамічна інформація про використання оперативної пам’яті
vmstat 1 – інформація про використання віртуальної пам’яті за заданий період часу
vmstat -d – статистика про введення-виведення жорсткого диску
slabtop – використання кеш пам’яті ядром
cat /proc/cpuinfo – інформація про процесор
cat /proc/meminfo – інформація про пам’ять
cat /proc/devices – інформація про всі пристрої
lsof | less – інформація про поточні відкриті файли і каталоги
dmesg | less – виведення інформації про кільцевий буфер ядра
lsmod – інформація про завантажені модулі
modinfo module – розширена інформація про конкретний модуль
ipcs –m – розподіл сторінок пам’яті
df – обсяги пам’яті на розділах
du – кількість дискових блоків в каталогах
hdparm /dev/sda – інформація про параметри жорсткого диску
lspci - інформація про pci пристрої
dmidecode – інформація про апаратні пристрої
netstat –s | less – статистика про пересилання мережевих пакетів
nmap 192.168.2.100 – сканування портів
lsdev – інформація про встановлені пристрої
free –m – інформація про вільну пам'ять (ОЗП і Swap) у Мбайтах

18 Мережа (LAN і WiFi)

ping 192.168.2.1 – перевірка доступності IP-адреси
arp –v – інформація з кешу ARP
ethtool eth0 – інформація про ethernet карти
export http_proxy=http://your.proxy:port - змінити значення змінної оточення http_proxy, для використання Інтернету через proxy-сервер
ifconfig eth0 — показати конфігурацію мережевого інтерфейсу eth0 (MAC адресу і IP-адресу TCP/IP з’єднання)
ifup eth0 — активувати (підняти) інтерфейс eth0
ifdown eth0 — деактивувати (опустити) інтерфейс eth0
ifconfig eth0 192.168.1.1 netmask 255.255.255.0 — виставити інтерфейсу eth0 IP-адресу і маску підмережі
ifconfig eth0 promisc — перевести інтерфейс eth0 в promiscuous-режим для «відловлення» пакетів (sniffing)
ifconfig eth0 -promisc — відключити promiscuous-режим на інтерфейсі eth0
dhclient eth0 — активувати інтерфейс eth0 в dhcp-режимі
route –n - вивести локальну таблицю маршрутизації
netstat -rn — вивести локальну таблицю маршрутизації
route add -net 0/0 gw IP_Gateway — задати IP-адресу шлюзу за замовчуванням (default gateway)
route add -net 192.168.0.0 netmask 255.255.0.0 gw 192.168.1.1 — добавити статичний маршрут в мережу 192.168.0.0/16 через шлюз с IP-адресою 192.168.1.1

route del 0/0 gw IP_gateway — вилучити IP-адресу шлюзу за замовчуванням (default gateway)
echo "1" > /proc/sys/net/ipv4/ip_forward — дозволити пересилання пакетів (forwarding)
hostname — відобразити мережене ім'я локальної машини
host itshaman.ru - виведення IP-адреси заданого сайту
host www.example.com або **host 192.0.43.10** — перетворення імені www.example.com хоста в IP-адресу і навпаки
ip addr show - отримання інформації про мережеві з'єднання
ip link show — відобразити стан усіх інтерфейсів
ip route - тестування шлюзу з таблиці маршрутизації
ip route show - виведення інформація про маршрутизацію
mii-tool eth0 — відобразити статус і тип з'єднання для інтерфейсу eth0
ethtool eth0 — відобразити статистику інтерфейсу eth0 з виведенням такої інформації, як підтримувані і поточні режими з'єднання
netstat –i - статистика про мережеві інтерфейси
netstat -an | grep LISTEN - список усіх відкритих портів
netstat -tupn — відобразити всі встановлені мережеві з'єднання за протоколами TCP і UDP без перетворення імен в IP-адреси і PID'и та імена процесів, які забезпечують ці з'єднання
netstat -tupln — відобразити всі мережеві з'єднання за протоколами TCP і UDP без перетворення імен в IP-адреси і PID'и і імена процесів, які слухають порти
tcpdump tcp port 80 — відобразити увесь трафік на TCP-порт 80 (звичайно — HTTP)
iwlist scan — просканувати ефір на предмет доступності бездротових точок доступу
iwconfig eth1 — показати конфігурацію бездротового мережевого інтерфейсу eth1
pppconfig - створення і налаштування Dial-Up з'єднання для виходу в Інтернет з використанням модему
pppoeconf - створення і налаштування виходу в Інтернет через ADSL-модем
service network status - інформація про мережу
wall Привіт - посилання повідомлення "Привіт" на термінали інших користувачів
lsof –i - список усіх відкритих портів у Інтернет
[sudo] netstat –tup - активні з'єднання з Інтернетом
socklist - виведення всіх відкритих сокетів
[sudo] netstat -anp --udp --tcp | grep LISTEN - список застосувань, які відкривають порти

19 Microsoft Windows networks(SAMBA)

nbtscan ip_addr - сканування IP-адрес
nmblookup -A ip_addr — дозволити netbios-імя nbtscan ставити за замовчуванням не у всіх системах. Можливо, прийдеться довстановлювати вручну. nmblookup включений у пакет samba.
smbclient -L ip_addr/hostname — відобразити ресурси, надані в загальний доступ на windows-машині
smbget -Rr smb://ip_addr/share — подібно до wget можна отримувати файли з windows-машин через smb-протокол
mount -t smbfs -o username=user,password=pass //winclient/share /mnt/share — змонтувати smb-ресурс, наявний на windows-машині, в локальну файлову систему

20 Керування міжмережевим екраном IPTABLES (firewall)

iptables -t filter –nL — відобразити ланцюжки правил
iptables -nL — відобразити всі ланцюжки правил
iptables -t nat -L — відобразити всі ланцюжки правил в NAT-таблиці
iptables -t filter -F або iptables -F — очистити всі ланцюжки правил в filter-таблиці
iptables -t nat -F — очистить всі ланцюжки правил в NAT-таблиці
iptables -t filter -X — вилучити всі користувацькі ланцюжки правил в filter-таблиці
iptables -t filter -A INPUT -p tcp --dport telnet -j ACCEPT — дозволити вхідне підключнння telnet'ом
iptables -t filter -A OUTPUT -p tcp --dport http -j DROP — блокувати вихідні HTTP-з'єднання

iptables -t filter -A FORWARD -p tcp --dport pop3 -j ACCEPT — дозволити «прокладати» (forward) POP3-з'єднання
iptables -t filter -A INPUT -j LOG --log-prefix "DROP INPUT" — включити журналювання ядром пакетів, які проходять через ланцюжок INPUT, і добавлення до повідомлення префікса «DROP INPUT»
ptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE — включити NAT (Network Address Translate) вихідних пакетів на інтерфейс eth0. Допускається при використанні з динамічно виділюваними IP-адресами.
iptables -t nat -A PREROUTING -d 192.168.0.1 -p tcp -m tcp --dport 22 -j DNAT --to-destination 10.0.0.2:22 — перенаправлення пакетів, адресованих одному хосту, на інший хост

21 Моніторинг і налагодження

at - запустити програми у визначений час
atq - виводить список завдань, поставлених в чергу командою at
atrm - вилучає завдання з черги команд at
/etc/crontab – файл, що містить таблицю розкладу запуску завдань
top — динамічно відобразити інформацію про запущені процеси, використовувані ними ресурси та іншу корисну інформацію (з автоматичним обновленням даних)
ps – інформація про процеси
ps, (ps –aux) - виводить інформацію про виконувані процеси користувачів
ps -eafw — відобразити запущені процеси, використовувані ними ресурси та іншу корисну інформацію (одноразово)
ps -e -o pid,args --forest — вивести PID'и і процеси як дерево
ps -x & - запуск фонового процесу . При завантаженні системи, специфічні процеси ("демонами"), завантажують у фоновий режим. Їх розміщують у каталозі /etc/rc.d/init.d/.
pstree — відобразити дерево процесів
pgrep - інформація про ID процесів
fuser – пошук процесів, які мають відкриті файли або сокети
nice - задати пріоритет процесу перед його запуском
renice - змінити пріоритет виконуваного процесу
kill – посилання сигналів процесу за PID процесу
killall - перервати виконання процесу за іменем процесу
kill -9 98989 или kill -KILL 98989 — «убити» процес з PID 98989 «на смерть» (без дотримання цілісності даних)
kill -TERM 98989 — коректно завершити процес з PID 98989
kill -1 98989 или kill -HUP 98989 — заставити процес з PID 98989 перечитати файл конфігурації
ipcs - взаємодія процесів (спільно використовувана пам’ять, семафори, повідомлення).Для отримання більш детальної інформації, можна використати help (наприклад: ps --help), або документацію (наприклад: man ps, для виходу натисніть q).
fg - вивести процес з фонового режиму
bg - продовжити виконання фонового процесу, якщо він зупинений натисканням клавіш <Ctrl+Z>
hash – інформація про hash таблиці
jobs – інформація про задачі
kill - завершити задачу
renice – змінити пріоритет фонової задачі, якою володіє користувач
lsof -p 98989 — відобразити список файлів, відкритих процесом з PID 98989
lsof /home/user1 — відобразити список відкритих файлів з каталогу /home/user1
strace -c ls >/dev/null — вивести список системних викликів, створених і отриманих процесом ls
strace -f -e open ls >/dev/null — вивести виклики бібліотек
watch -n1 'cat /proc/interrupts' — відображати переривання в режимі реального часу
last reboot — відобразити історію перевантажень системи
last user1 — відобразити історію реєстрації користувачів user1 у системі і час його знаходження у ній
lsmod — вивести завантажувані модулі ядра
free -m — показати стан оперативної пам'яті в мегабайтах
smartctl -A /dev/hda — контроль стану жорстокого диску /dev/hda через SMART

smartctl -i /dev/hda — перевірити доступність SMART на жорсткому диску /dev/hda
tail /var/log/dmesg — вивести десять останніх записів з журналу завантаження ядра
tail /var/log/messages — вивести десять останніх записів з системного журналу

Комбінації клавіш для керування завданнями:

<Ctrl+Z> - призупинити виконання завдань
<Ctrl+C> - завершити виконання завдань

22 Каталоги файлової системи Linux

Кореневий каталог "/" ОС Linux містить підкаталоги із стандартними іменами, які визначаються стандартом (Filesystem Hierarchy Standard).

Підкаталоги кореневого каталогу

Каталог	Опис
/bin	Назва цього каталогу походить від слова "binaries". У цьому каталозі знаходяться файли, самих необхідних утиліт. Сюди попадають такі програми, які можуть знадобитися системному адміністраторові або іншим користувачам для усунення неполадок у системі або при відновленні після збою.
/boot	"boot" - завантаження системи. У цьому каталозі знаходяться файли, необхідні для найпершого етапу - завантаження ядра і, звичайно, саме ядро. Користувачеві практично ніколи не потрібно безпосередньо працювати із цими файлами.
/dev	У цьому каталозі знаходяться всі наявні в системі файли призначені для роботи з різними системними ресурсами і пристроями (англ. "devices" - "пристрої", звідси й скорочена назва каталогу). Наприклад, файли /dev/ttyN відповідають віртуальним консолям, де N - номер віртуальної консолі. Дані, введені користувачем на першій віртуальній консолі, система зчитує з файлу /dev/tty1; у цей же файл записуються дані, які потрібно вивести користувачеві на цю консоль.
/etc	Каталог для системних конфігураційних файлів. Тут зберігається інформація про специфічні налаштування даної системи: інформація про зареєстрованих користувачів, доступні ресурси, налаштування різних програм.
/home	Тут розташовані каталоги, що належать користувачам системи - домашні каталоги, звідси й назва "home". Відокремлення всіх файлів, що створюються користувачами, від інших системних файлів дає очевидну перевагу: серйозне ушкодження системи або необхідність відновлення не торкнеться найціннішої інформації - файлів користувача.
/lib	Назва цього каталогу - скорочення від "libraries" (англ. "бібліотеки"). Бібліотеки - це набір стандартних функцій, необхідних багатьом програмам: операцій введення/виведення, виведення елементів графічного інтерфейсу і т.і. Щоб не включати ці функції в текст кожної програми, використовуються стандартні функції бібліотек - це значно заощаджує місце на диску й спрощує написання програм. У цьому каталозі знаходяться бібліотеки, необхідні для роботи найбільш важливих системних утиліт (розміщених в /bin і /sbin).

/media	Каталог для монтування (від англ. "mount") - тимчасового підключення файлових систем, наприклад, на знімних носіях (CD-ROM та інші.).
/proc	У цьому каталозі всі файли "віртуальні" - вони розташовуються не на диску, а в оперативній пам'яті. У цих файлах знаходиться інформація про програми (процеси), що виконуються у даний момент у системі.
/root	Домашній каталог адміністратора системи - користувача root. Розміщення його окремо від домашніх каталогів інших користувачів необхідно тому, що /home може розташовуватися на окремому пристрої, який не завжди доступний (наприклад, на мережевому диску), а домашній каталог root повинен бути присутнім у будь-якій ситуації.
/sbin	Каталог для найважливіших системних утиліт (назва каталогу – скорочення від "system binaries"): на додаток до утиліт /bin тут знаходяться програми, необхідні для завантаження, резервного копіювання, відновлення системи. Повноваження на виконання цих програм є тільки в системного адміністратора.
/tmp	Цей каталог призначений для тимчасових файлів: у таких файлах програми зберігають необхідні для роботи проміжні дані. Після завершення роботи програми тимчасові файли втрачають сенс і повинні бути вилучені. Як правило, каталог /tmp очищається при кожному завантаженні системи.
/usr	Каталог /usr- це "держава в державі". Тут можна знайти такі ж підкаталоги bin, etc, lib, sbin, як і в кореневому каталозі. Однак у кореневий каталог попадають тільки утиліти, необхідні для завантаження й відновлення системи в аварійній ситуації, а всі інші програми й дані розташовуються в підкаталогах /usr. Прикладних програм у сучасних системах звичайно встановлено дуже багато, тому цей розділ файлової системи може бути дуже великим.
/var	Назва цього каталогу - скорочення від "variable" ("змінні" дані). Тут розміщаються ті дані, які створюються в процесі роботи різними програмами й призначені для передачі іншим програмам і системам (черги друку, електронної пошти та ін.) або для інформування системного адміністратора (системні журнали, що містять протоколи роботи системи). На відміну від каталогу /tmp сюди попадають ті дані, які можуть знадобитися після того, як програма, що їх створила, завершить роботу.

Висновок.

• ОС Linux має великий набір команд (понад 530). Команди Linux за функціональним призначення поділяються на групи.

• Команди Linux використовуються у випадках коли спрацьовують функції графічного інтерфейсу, при віддаленому адмініструванні серверів, для реалізації функцій, які не забезпечує графічне середовище, якщо не стартує або пошкоджене графічне середовище X Window.

• Для організації обчислювальних процесів можна використати команди наступних груп:

- службові інформаційні команди (інформація про команди і їх пошук);

- адміністративні задачі;
- файли і каталоги;
- запуск завданнями, керування процесами (сигнали);
- використання ресурсів і пристроїв;
- мережеві.

Література.

1. С.Л. Скловская. Команды Linux. – СПб.: Питер, 2004. – 848 с.

2. Митчелл М., Оулдем Д., Самьюєл А. Программирование для Linux. Профессиональный поход. – М.: Вильямс, 2002. – 288 с.

Запитання.

1. Для чого призначені команди ОС?
2. Які команди використовуються для отримання загальної довідкової інформації?
3. Які команди використовуються для пошуку окремої команди?
4. Які команди використовуються для адміністрування ОС?
5. Які команди використовуються для роботи з файлами і каталогами?
6. Як створюються м’які і жорсткі посилання на файли і в чому між ними різниця?
7. Як є команди для роботи із стеком каталогів?
8. Які команди використовуються для роботи із завданнями, процесами і сигналами?
9. За допомогою яких команд можна отримати інформацію про використання ресурсів і пристроїв.
10. Засоби для встановлення програмних пакетів.
11. За допомогою яких команд можна отримати інформацію про мережеві з’єднання.
12. Основні каталоги і підкаталоги ОС Linux, їх призначення.

8. МОВА СЦЕНАРІЇВ BASH

Мета. Вивчення базових можливостей оболонки і сценаріїв Bash

Вступ. Подібно до інших оболонок, доступних в Linux, Bash (Bourne Again shell) є не тільки командною оболонкою, але і мовою написання сценаріїв (скриптів). Сценарії дозволяють в повній мірі використати можливості оболонки і автоматизувати багато задач, які потребують для свого виконання введення багатьох команд. Крім Bash використовуються і інші мови сценаріїв, такі як Perl, Lisp, Tcl, Python, Ruby.

План.

1 Базові можливості оболонки і сценаріїв Bash
1.1 Ключові слова Bash
1.2 Запуск оболонки Bash і виконання сценаріїв
1.3 Секції Bash-сценарію
1.4 Команди Linux і команди оболонки Bash
2 Змінні Bash
2.1 Зарезервовані змінні середовища Bash
3 Інструкції умов
3.1 Команда `test`
3.2 Подвійні квадратні дужки
3.3 Подвійні круглі дужки
3.4 Інструкція `case`
4 Арифметичні вирази
4.1 Додаткові можливості виконання арифметичних виразів
5 Робота із стрічками
6 Регулярні вирази
7 Розширення імен файлів у Bash
8 Інструкції для організації циклів
9 Службові символи, які використовуються у сценаріях Bash

1 Базові можливості оболонки і сценаріїв Bash

Оболонка (shell) це програма, яка виконує команди ОС. Перша UNIX оболонка була написана Steven R. Bourne у 1974 році. Вдосконалена версія оболонки була розроблена як частина GNU проекту, який використовувався в ОС Linux. Ця оболонка була названа Bash (“Bourne Again Shell”).

Сценарії це невеликі і прості програми, які використовують ключові слова оболонки для керування виконанням команд ОС. Сценарії виконуються інтерпретаторами порядково і тому швидкість виконання їх невисока.

1.1 Ключові слова Bash

```
!        case     Do       done     elif     else
esac     fi       For      function if       in
select   then     Until    while    time     {
}        [[       ]]
```

1.2 Запуск оболонки Bash і виконання сценаріїв

Оболонка Bash запускається командою `sh`:

```
->sh
```

```
sh-4.2$
```

В запущеній оболонці Bash сценарій (послідовність команд) може виконуватися в інтерактивному режимі. В командному рядку можна вводити одну або декілька команд

```
$ date
$ who
$ date; who
```

Якщо команди не поміщаються в командному рядку, то їх можна продовжити в наступному рядку використовуючи символ ”\”.

Команди можна згрупувати використовуючи фігурні дужки. Результат роботи однієї команди можна передати в іншу використовуючи символ конвеєру ”|”. Наприклад, результат виконання команди `ls` стає входом для команд у фігурних дужках:

```
$ ls -l | { while read FILE do ; echo "$FILE" done }
```

Команди можна об’єднувати з використанням булевих виразів. Приклад перевірки наявності файлу `orders.txt` та виведення про нього інформації з наступним вилученням:

```
$ test -f orders.txt && { ls -l orders.txt ; rm orders.txt; } \
|| printf "no such file"
```

Вихід з оболонки Bash здійснюється командою `exit`:

```
$ exit
exit
```

Сценарій Bash може виконуватися у пакетному режимі у Linux консолі. Для цього сценарій Bash записується з використанням простого текстового редактора (kwrite, vim) і зберігається у файлі з розширенням `.sh`, наприклад:

```
# hello.sh
# This is my first shell script
printf ”Привіт Bash!\n”
exit 0
```

Виконати сценарій можна запуском нової Bash оболонки

```
$ bash hello.sh
```

або запустити як бінарний файл у Linux консолі

```
~> chmod +x hello.sh
~> ./hello.sh
Привіт Bash!
```

Виконати сценарій у *режимі налагодження* можна запуском Bash з параметром `-x`

```
$ bash –x hello.sh
```

1.3 Секції Bash-сценарію

При написанні Bash сценарію рекомендується використовувати наступні секції:
- заголовок;
- глобальні оголошення;
- перевірка наявності необхідних команд і файлів;
- основний сценарій;
- завершення.

В заголовку вказується назву використовуваного сценарію і задаються опції на виконання оболонки:

```
#!/bin/bash
# коментарі
shopt -s -o nounset # виявити невизначені змінні
shopt -o errexit # завершити сценарій, якщо команда завершилася з помилкою
```

```
shopt –o xtrace  # вивести на консоль кожну команду перед її виконанням
```

Глобальні оголошення – оголошують змінні, які доступні у всьому сценарії і його функціях:

```
# Global Declarations
declare -rx SCRIPT=${0##*/} # SCRIPT є іменем цього сценарію
declare -rx who="/usr/bin/who" # команда who  - man 1 who
declare -rx sync="/bin/sync"   # команда sync - man 1 sync
declare -rx wc="/usr/bin/wc"   # команда wc   - man 1 wc
```

Наявність необхідних команд і файлів перевіряють перед виконанням основного сценарію.

```
if test -z "$BASH" ; then
printf "$SCRIPT:$LINENO: сценарій виконується в BASH оболонці\n" >&2
exit 192
fi
if test ! -x "$who" ; then
printf "$SCRIPT:$LINENO: команда $who недоступна — \
aborting\n " >&2
exit 192
fi
if test ! -x "$sync" ; then
printf "$SCRIPT:$LINENO: команда $sync недоступна — \
aborting\n " >&2
exit 192
fi
if test ! -x "$wc" ; then
printf "$SCRIPT:$LINENO: команда $wc недоступна — \
aborting\n " >&2
exit 192
fi
```

Основний сценарій виконує функціональне призначення всього сценарію, наприклад

```
# Примусовий запис змінених блоків на диск при відсутності користувачів
USERS=`who | wc -l`
if [ $USERS -eq 0 ] ; then
sync
fi
```

При завершенні сценарію очищаються, при наявності, тимчасові файли і повертається код завершення сценарію:

```
exit 0   # нормальне завершення
```

1.4 Команди Linux і команди оболонки Bash

Для забезпечення незалежності оболонки Bash від різних версій Linux в ній реалізовано (`builtin`) частину базових команд Linux. Перелік реалізованих команд оболонки описаний в довідковій системі:

```
$ man builtin
```

Визначити, чи команда належить оболонці Bash або Linux можна командою **`type`**.

```
$ type cd
cd is a shell builtin  # команда вбудована в оболонку Bash
$ type id
id is /usr/bin/id      # команда Linux
```

Вивести всі варіанти належності команди дозволяє ключ –a:

```
$ type -a pwd
pwd is a shell builtin  # команда вбудована в оболонку Bash
```

```
pwd is /bin/pwd.        # команда Linux
```

Виконати вбудовану в оболонку команду

```
builtin pwd
```

Виконати Linux команду

```
command pwd
```

Команда `enable` керує доступом до вбудованих в оболонку Bash команд

```
$ enable test      # відкриття доступу до builtin
$ type test
test is a shell builtin
$ enable -n test   # закриття доступу до builtin
$ type test
test is /usr/bin/test
```

Рекомендується використовувати команду `enable` у секції глобальних оголошень, так як досить складно буде перевіряти у сценарії статус кожної команди на приналежність до вбудованих в оболонку Bash.

В одному рядка можна вводити *одну* або *декілька* команд:

```
$ date
$ pwd
$ date; pwd   # після останньої команди не ставиться «;».
```

Команди Bash можна об'єднувати у *групу* (після останньої команди ставиться «;»), яка задається фігурними дужками:

```
$ { sleep 5 ; printf ″%s\n″ ″Slept for 5 seconds″ ; }
Slept for 5 seconds
```

Підоболонка (subshell) – це набір команд у круглих дужках, які повертають один код завершення і мають свої змінні середовища:

```
$ ( sleep 5 ; printf ″%s\n″ ″Slept for 5 seconds″ )  # підоболонка
Slept for 5 seconds

$ declare -ix COUNT=15
$ { COUNT=10 ; printf ″%d\n″ ″$COUNT″ ; }  # група
10

$ printf ″%d\n″ ″$COUNT″
10
$ ( COUNT=20 ; printf ″%d\n″ ″$COUNT″ ) # підоболонка
20

$ printf ″%d\n″ ″$COUNT″
10
```

Підоболонки часто використовуються з каналами

```
# subshell.sh
#!/bin/bash
# Виконання деякої операції з усіма файлами каталогу
shopt -s -o nounset
declare -rx SCRIPT=${0##*/} # ім'я сценарію
declare -rx INCOMING_DIRECTORY=″incoming″
ls -1 ″$INCOMING_DIRECTORY″ |
(
  while read FILE ; do
  printf ″$SCRIPT: Processing %s...\n″ ″$FILE″
  # <-- деяка операція над файлом
  done
)
printf ″Файли оброблено\n″
exit 0
```

2 Змінні Bash

Усі змінні Bash, які використовуються у сценарії або функціях є за замовчуванням глобальними. Змінні оголошені командою `local` є локальними в межах функції або блоку де вони оголошені.

```
$ local loc_var=23
```

У Bash змінні не розділяються за типами. Усі Bash змінні зберігаються як стрічкисимволів, але у залежності від того, який зміст зберігається у змінних, їх можна поділити на наступні типи:

- стрічкові змінні;
- цілочисельні змінні;
- змінні константи;
- змінні масиви.

Присвоїти значення змінній можна за допомогою інструкції ”=” (без пропусків перед і після) або команди `let`. Приклад присвоєння змінній значень цілочисельного, дійсного і стрічкового:

```
$ a=1; b=2.0; c=3e-4; d=”Привіт Bash”
$ echo $a $b $c $d
1 2.0 3e-4 Привіт Bash
```

Присвоєння значень арифметичних виразів з використанням команди **`let`**

```
$ let i=a+1; echo $i
2
```

Неініціалізована змінна має **`”null”`** значення (не нуль).

```
$ if [ -z "$unassigned" ]; then echo "\$unassigned is NULL" ; fi
unassigned is NULL
```

Підставлення змінних. Ім’я змінної задає місце для зберігання значення. Для отримання значення змінної `var` використовують посилання на її значення `$var`. Посилання `$var` є спрощеною формою від загальної `${var}`, яка дозволяє однозначно виділити ім’я змінної у стрічках.

```
$ var=23
$ echo var
var
$ echo $var
23
$ echo ${var}
23
$ printf ”%s\n” ”Вартість ${var} грв”
Вартість 23 грв
```

Заміна частини значення змінної ‘33’ на ‘dd’

```
$ var=2334
$ b=${var/33/dd}
$ echo $b
2dd4
```

Непрямі посилання на змінні.

Якщо одна змінна містить ім’я другої змінної, то можна отримати значення другої змінної через звернення до першої:

```
$ a=b;b=”hello”
$ eval a=\$$a
$ echo “a=$a”
a=hello
```

Команда declare (typeset).

Команда `declare` і `typeset` є синонімами і призначені для накладення обмежень на змінні.

Змінні Bash мають **атрибути**, які можуть бути ввімкнені (-) або вимкнені (+) за допомогою команди `declare`.

Команда `declare` робить змінні локальними. За її допомогою можна створити асоціативні масиви, цілочисельні змінні і змінні тільки для читання, а так задати інші обмеження.

Масив: `declare` (синонім `typeset`) `-a` *`variable`* `-` змінна розпізнається оболонкою як асоціативний масив (індексований список змінних).

```
# індексація починається з 1
$ declare –a array
$ array=( [1]=one [2]=two )  # або array1=( [1]=one [2]=two )
$ echo array ${array[@]}     $ echo array1 ${array[@]}
one two                      one two

# Продовження елементів масиву у новому рядку
$ declare -a DEPT[0]=″замовлення″ DEPT [1]=″доставлення″ \
  DEPT [2]=″обслуговування замовників″
$ echo ″${DEPT[0]}″
замовлення
$ echo ″${DEPT[2]}″
обслуговування замовників
$ printf “%s\n″ “${DEPT[*]}″
замовлення доставлення обслуговування замовників

# Створення масиву з використанням круглих дужок. Індексація починається з 0
$ array2=(zero one two)
$ echo ${array2[@]}
zero one two
$ echo ${array2[1]}
one

$ MAS=(“North America″ “Europe″ “Far East”)
$ printf “%s\n″ “${MAS[*]}″
North America Europe Far East

# Створення масиву з використанням круглих дужок і індексів:
$ MAS=([3]=″North America″ [2]=″Europe″ [1]=″Far East”)
$ printf “%s\n″ “${MAS[*]}″
Far East Europe North America
```

Елементами масиву можуть бути змінні

```
$ a=1;b=2;c=3
$ array=($a,$b,$c)
$ echo ${array[@]}
1 2 3
```

Вилучення елемента массива

```
$ array2=(zero one two)
$ unset array2[1]
$ echo ${array2[@]}
zero two
```

Вилучення масиву

```
$ unset array2
$ echo ${array2[@]}
```

Добавлення елементу в кінець масива

```
$ array2=(zero one two)
$ array2+=(three)
$ echo ${array2[@]}
```

```
zero one two three
```

Добавлення елементу в задану позицію

```
$ array2=(zero one two)
$ array2+=([1]=three)
$ echo ${array2[@]}
zero three one two
```

Добавлення елементу, який є сумою двох інших елементі

```
$ array3=(1 2 3)
$ array3[3]=`expr ${array3[1]} + ${array3[2]}
$ echo ${array3[@]}
1 2 3 5
```

Число елементів масиву:

```
$ echo "${#array3[*]}"
4
```

Отримання елементів масиву із другої позиції

```
$ array3=(1 2 3 5)
$ echo ${array2[@]:2}
3 5
```

Отримання двох елементів масиву із першої позиції

```
$ array3=(1 2 3 5)
$ echo ${array2[@]:1:2}
2 3
```

Заповнити елементи масиву із файлу

```
$ array= $( `cat "1.sh"` )
```

Створення масиву з іменами файлів поточного каталогу:

```
$ file=(*)
$ echo "${file[@]}"   # або "${file[*]}"
file1 file2 file3
```

Імена функцій: `declare –f <імена функцій>`
Виводить раніше оголошені функції.

Ціле: `declare -i` *`variable`* (змінна розпізнається оболонкою як ціле значення. Присвоєння значення цій змінній автоматично вмикає систему арифметичного оцінювання).

```
$ declare -i a=5+2; echo $a; unset a
7
$ declare +i a
```

Символи нижнього регістру: `declare -l` *`variable`* (всі символи змінної конвертуються у нижній регістр).

Тільки читання: `declare -r` *`variable`* (змінна розпізнається оболонкою як тільки для читання, її не можна буде змінити або знищити командою `unset`).

```
$ declare -r COMPANY="Старе місто"
$ printf "%s\n" "$COMPANY"
$ COMPANY="Нове місто"
```

Трасування: `declare -t` *`variable`* (присвоєння змінній атрибуту `trace`).

Символи верхнього регістру: `declare -u` *`variable`* (всі символи змінної конвертуються у верхній регістр).

Змінні оболонки Bash існують в сценарії або інтерактивній сесії в яких вони оголошені. Для того, щоб вони були доступними в інших місцях їх потрібно експортувати.

Експорт: `declare -x` *`variable`* (змінна для експорту успадковується любою підоболонкою обо дочірним процесом).

```
$ declare -x CVSROOT="/home/cvs/cvsroot"
```

Створені змінні існують до завершення сценарію або знищення їх вбудованою командою `unset`

```
$ unset COST
```

Результат команди може бути присвоєний змінній, якщо команда взята у ліво нахилені апострофи (`) або у `$(…)`.

```
$ declare NUMBER_OF_FILES
$ NUMBER_OF_FILES=`ls -l | wc -l`
# NUMBER_OF_FILES=$(ls -l | wc -l)
$ printf "%d" "$NUMBER_OF_FILES"
14
```

2.1 Зарезервовані змінні середовища Bash

Bash має більш як 50 зарезервованих змінних середовища. Ці змінні створюються при першому запуску оболонки Bash, забезпечують інформацію про поточну сесію і можуть використовуватися для контролю деяких параметрів оболонки. Імена змінних користувача не повинні співпадати з іменами зарезервованими змінними.

Команда `declare` без параметрів *виводить значення всіх зарезервованих змінних*. Приклад значень деяких з них:

```
$BASH – повний шлях до оболонки Bash
$BASH_VERSION - версія Bash (наприклад 4.2.45(1)-release)
$COLUMNS – число символів в рядку екрана
$HOSTNAME – мережеве ім’я комп’ютера
$HOSTTYPE – тип комп’ютера
$HOME – домашній каталог
$IFS - список символів, які використовуються для розбиття рядків на слова в
      оболонці
$LINENO – поточний номер рядка у сценарії або функції
$LINES – число горизонтальних рядків екрану
$MACHTYPE – апаратна архітектура
$OSTYPE – назва ОС
$PATH – список шляхів пошуку команд для виконання
$PPID – ID процесу батьківського до процесу оболонки
$PS1 – стрічка основного повідомлення командного рядка
$PS2 - стрічка вторинного повідомлення командного рядка
$PS3 - стрічка третинного повідомлення командного рядка
$PS4 - повідомлення на початку кожної стрічки для команди trace або запуску
       сценарію з ключем -x
$PWD – поточний робочий каталог
$RANDOM – випадкове число між 0 і 32767
$SECONDS – час роботи сценарію в секундах
$SHELL – вибрана оболонка
$TERM – тип емуляції терміналу
$UID – ідентифікатор користувача в /etc/password
```

3 Інструкції умов

В інструкціях умов використовуються значення істина (`true`) і фальш (`false`):

```
$ true
$ printf "%d\n" "$?"
0
$ false
$ printf "%d\n" "$?"
```

```
1
```

Інструкція `if-then` виконує команду/команди оболонки Bash або Linux і перевіряє їх результат завершення. Якщо код завершення команди або послідовності команд є не нульовим, то здійснюється перехід до наступної команди.

```
if command1; … commandN        if command; … commandN; then
then                              commands
  commands                     fi
fi
```

```
$ if date
$ then
$   echo "це працює"
$ fi
Це працює
```

```
$ if date; then echo "це працює"; fi
Це працює
```

Для оброблення альтернативної команди використовується інструкція `if-then-else`.

```
if command                 if command; then
then                        commands
   commands                else commands
else                       fi
   commands
fi                         if command; then commands; else commands; fi
```

```
if команда1; команда2
then echo ”Всі команди виконано”
else echo ”Не всі команди виконано”
fi
```

```
if  cmp a b  &> /dev/nul
then echo ”файли ідентичні”
else echo ”файли різні”
fi
```

Інструкція `if-then-else` може мати вкладені перевірки умов.

```
if command1                if command1; then
then                         command set 1
  command set 1            elif command2; then
elif command2                command set 2
then                       elif command3; then
  command set 2               command set 3
elif command3              elif command4 ; then
then                          command set 4
   command set 3           fi
elif command4
then
   command set 4
fi
```

3.1 Команда test

Команда `test` – це вбудована команда Bash, яка дозволяє перевіряти не *тільки результати завершення команд*, але і *результати порівняння різних виразів*. Аргументами команди `test` можуть бути вирази порівняння чисел, символьних рядків або файлів. Команда `test` повертає у відповідності з результатами перевірки істину (0) або фальш (1).

Синтаксис команди: `test умова`. Команда може об’єднуватися з `if-then` інструкцією.

```
if test умова
then команди
```

```
fi
```

Команди можна *згрупувати використовуючи фігурні д*ужки. Так перевіряється наявність файлу `orders.txt`, якщо він існує, то виводиться про нього інформація і файл вилучається:

```
$ test -f orders.txt && { ls -l orders.txt ; rm orders.txt; } \
```

Bash використовує як синонім для команди `test` одинарні квадратні дужки […] (в реалізації тільки ліву дужку)

```
$ type [
  - is a shell built-in

if [ умова ]
then команди
fi
```

Команда `test` може оцінювати три класи умов:

- **порівняння числових виразів**, `[ n1 -x n2 ]`, де `x`:

eq (= =), **ge** (>=), **gt** (>), **le** (<=), **lt** (<), **ne** (!=);

```
n1 -eq n2 — (однакові) True якщо n1 дорівнює n2
n1 -ge n2 — True якщо n1 є більше або рівне n2
n1 -gt n2 — True якщо n1 є більше n2
n1 -le n2 — True якщо n1 є менше або рівне n2
n1 -lt n2 — True якщо n1 є менше n2
n1 -ne n2 — True якщо n1 не дорівнює n2
if [ $val1 -gt 5 ]
```

- **порівняння стрічок**, `[ str1 -x str2 ]`, де `x`:

=, **!**=, <, >, **n** (довжина більше нуля), **z** (довжина нуль);

(при порівнянні стрічок з використанням символів “>”, “<” їх необхідно екранувати (символ “\”), щоб вони не сприймалися як символи перенаправлення).

```
-z str — (zero length) True якщо стрічка є пуста
```

```
#!/bin/bash
a1=Testing
a2=testing
# if [ $a1 \> $a2 ]; then echo "$a1 > $a2"; else echo "$a1 < $a2"; fi
if [ $a1 \> $a2 ]
then
   echo "$a1 > $a2"
else
   echo "$a1 < $a2"
fi
$ ./test
Testing є менше ніж testing
```

- **порівняння файлів**, `[ -x file ]`, де:

```
-b file — True якщо файл є блоковим пристроєм
-c file — True якщо файл є символьним пристроєм
-d file — True якщо файл є каталогом
-e file — True якщо файл існує
f1 -ef f2 — (еквівалентні файли) True якщо файл f1 є жорстким посиланням на файл f2
-f file — True якщо файл існує і є дійсно файлом
-g file — True якщо файл має встановлені права доступу групи
-G file — True якщо файлом володіє ваша група
-h file (or -L file) — True якщо файл є символічним посиланням
-k file — True якщо файл має встановлений sticky біт доступу
-n s (or just s) — (not null) якщо стрічка не порожня
-N file — True якщо файл має новий вміст (з часу останньої операції read)
f1 -nt f2 — True якщо файл f1 новіший від файла f2
```

-O *file* — True якщо Ви є (ефективним) власником файлу
f1 **-ot** *f2* — (*older than*) True якщо файл *f1* старіший від файлу *f2*
-p *file* — True якщо файл є каналом (pipe)
-r *file* — True якщо файл можна читати (Вашим сценарієм)
-s *file* — True якщо файл існує і не пустий
-S *file* — True якщо файл є сокет
-t fd — True якщо файловий дескриптор відкритий в терміналі
-u *file* — True якщо файл має встановлені права доступу користувача
-w *file* — True якщо у файл можна записати (Вашим сценарієм)
-x *file* — True якщо файл можна виконати (Вашим сценарієм)

Для створення складних умов перевірки в команді `test` застосовується булева логіка:

```
[ умова1 ] && [ умова 2 ]
[ умова 1 ] || [ умова 2 ]
#!/bin/bash
if [ -d $HOME ] && [ -w $HOME/testing ]
then
  echo "Файл існує і в нього можна записати"
else
  echo "У файл не можна записати"
fi
```

3.2 Подвійні квадратні дужки

Варіант команди `test` як подвійні квадратні дужки `[[ вираз ]]` забезпечує *розширені умови порівняння символьних стрічок*, наприклад порівняння з використанням регулярних виразів (шаблонів). Приклад перевірки значення змінної середовища `$USER`, чи воно починається з букви "`R`":

```
if [[ $USER == R* ]] …
```

3.3 Подвійні круглі дужки

Подвійні круглі дужки `(( вираз ))` (синонім команди `let`) дозволяють *обчислювати і порівнювати в умовах складні арифметичні вирази*. У виразі можна порівнювати арифметичні значення як із стандартними (команди test), так і додатковими арифметичними операціями: **val++**, **val--**, **++val**, **--val**, **!** (логічна інверсія), **~** (бітова інверсія), ** (експонента), **<<**, **>>**, **&** (бітове AND), **|** (бітове OR), **&&** (логічне AND), **||** (логічне OR).

Вирази у подвійних круглих дужках можна використовувати не тільки в інструкції `if`, але і як звичайну команду в сценарії для присвоєння значень змінним.

```
#!/bin/bash
val1=10
if (( $val1 ** 2 > 90 ))
then
    # let val2 = $val1 ** 2
   (( val2 = $val1 ** 2 ))
echo "Квадрат $val1 дорівнює $val2"
fi
$ ./test
The Квадрат 10 дорівнює 100
```

3.4 Інструкція case

Для вибору значення однієї змінної з багатьох можливих значень (замість інструкції `if-then-else`) використовується інструкція `case`:

```
case variable in
pattern1 | pattern2) commands1;;
pattern3) commands2;;
*) default commands;;
esac
```

```
#!/bin/bash
USER="Петро"
case $USER in
Іван)
echo "Привіт $USER";;
Петро)
echo "$USER у вас обмежений доступ";;
*)
echo "Вхід заборонений";;
esac
```

4 Арифметичні вирази

Арифметичні вирази обчислює вбудована команда `let`. Команда `let` сприймає стрічку, яка містить ім'я змінної, знак дорівнює і вираз для обчислення. Результат обчислення присвоюється змінній.

```
$ let "SUM=5+5"
$ printf "%d" $SUM
10
```

У виразі для обчислень команди `let` можна не використовувати знак видобування значення змінної `$`.

```
$ let "SUM=SUM+5"
$ printf "%d" "$SUM"
15
$ let "SUM=$SUM+5"
$ printf "%d" "$SUM"
20
```

Якщо змінна декларована як `integer` (ключ –і), то `let` команда виконується за замовчуванням:

```
$ declare -i SUM
$ SUM=SUM+5
$ printf "%d\n" $SUM
25
```

Якщо змінна оголошена як стрічка (за замовчування всі змінні є стрічковими), то їй будуть назначатися стрічкове значення

```
$ unset SUM      # знищення змінної
$ declare SUM=0 # оголошення і ініціалізація глобальної змінної
$ SUM=SUM+5      # присвоєння нового стрічкового значення
$ printf "%s\n" $SUM
SUM+5
```

Приклад округлення до найближчого 10

```
$ declare -i COST=5234
$ COST=\(COST+5\)\10*10  # екранування круглих дужок
$ printf "%d\n" $COST
5230
```

Команда `let` дозволяє виконувати наступні арифметичні операції:

```
-, + – унарний мінус і плюс
!, ~ – логічну і бітову інверсію
*, /, % – ділення, множення, залишок від ділення
+, - – додавання, віднімання
```

```
<<, >> — лівий, правий бітовий зсув
<=, >=, <, > - порівняння
==, != — рівність, нерівність
& — побітове І (AND)
^ — побітове виключне АБО (XOR)
| — побітове АБО (OR)
&& — логічне І (AND)
|| — логічне АБО (OR)
expr ? expr1 : expr2 - інструкція умови
=, *=, /=, %= — присвоєння
+=, -=, <<=, >>=, &=, ^=, |= — операції самопосилання
```

В команді `let` логічна істина (`true`) задається як 1, а логічна фальш (`false`) як 0. Любе значення відмінне від 1 і 0 також розглядається як істина. Логічна істина це не те саме, що і успішне завершення команди (код повернення 0), яке перевіряє команда `test`. Тому значення true в командах `let` і `test` протилежні.

```
$ let "RESULT=1 > 0"
$ printf "1 більше 0 : %d\n" "$RESULT"
1 більше 0 : 1
```

Команда `let` може обробляти вісімкові і шістнадцяткові значення

```
$ declare -i OCTAL=0 HEX=0
$ let "OCTAL=0775"      # вісімкове значення
$ let "HEX=0x1F"        # шістнадцяткове значення
$ echo "$OCTAL $HEX"    # виведення десяткових значень
509 31
```

Команда `let` має синонім – подвійні круглі дужки ((…)). Вони використовуються для того, щоб вбудовувати `let` вираз як параметр у інші команди.

```
$ declare -i X=5
$ while (( X-- > 0 )) ; do
> printf "%d " "$X"
> done
4 3 2 1 0
```

В подвійних круглих дужках можуть використовуватися наступні операції: **`var++`**, **`var--`**, **`++var`**, **`--var`**, **`!`** (логічна інверсія), **`~`** (побітова інверсія), **`**`** (експонента), **`<<`**, **`>>`** (зсув бітів вліво, вправо), **`&`** (бітове І), **`|`** (бітове АБО), **`&&`** (логічне І), **`||`** (логічне АБО).

```
$ echo $(( $var1 ** 2 > 30 ))
1
$ echo $(( 2**8 ))
256
```

4.1 Додаткові можливості виконання арифметичних виразів

До додаткових можливостей виконання арифметичних виразів у Bash відносяться:

- використання команди `expr`;
- використання конструкції `$[ вираз ]`;
- використання вбудованого калькулятора `bc` для виконання операцій з плаваючою крапкою.

Команда `expr` розпізнає як математичні, так і стрічкові інструкції.

Таблиця 1.1 – **Операції команди `expr`**

ARG1 < ARG2	повертає 1, якщо arg1<arg2, інакше 0
ARG1 <= ARG2	повертає 1, якщо arg1<=arg2, інакше 0
ARG1 = ARG2	повертає 1, якщо arg1=arg2, інакше 0
ARG1 != ARG2	повертає 1, якщо arg1!=arg2, інакше 0
ARG1 >= ARG2	повертає 1, якщо arg1>=arg2, інакше 0

ARG1 > *ARG2*	повертає 1, якщо arg1>arg2, інакше 0
ARG1 + *ARG2*	повертає суму arg1,arg2
ARG1 - *ARG2*	повертає різницю arg1,arg2
ARG1 * *ARG2*	повертає добуток arg1,arg2
ARG1 / *ARG2*	повертає частку від ділення arg1,arg2
ARG1 % *ARG2*	повертає залишок від ділення arg1,arg2
STRING : *REGEXP*	повертає стрічку, яка відповідає шаблону
match *STRING REGEXP*	повертає стрічку, яка відповідає шаблону
substr *STRING POS LENGTH*	повертає стрічку довжини LENGTH з позиції POS
index *STRING CHARS*	повертає позиції CHARS у стрічці STRING
length *STRING*	повертає довжину стрічки
(*EXPRESSION*)	повертає значення EXPRESSION

Для присвоєння значення обчисленого арифметичного виразу змінній, використовуються лівонахилені апострофи.

```
#!/bin/bash
var1=10
var2=20
# var3=$(( $var2 / $var1 ))
var3=`expr $var2 / $var1`
echo Результат $var3
```

Екранування символу ”*” в команді `expr`:

```
$ expr 5 \* 2
10
```

В Bash присвоїти значення арифметичного виразу змінній можна за допомогою конструкції `$[вираз]`:

```
$ var1=$[1 + 5]
$ echo $var1
6
$ var2 = $[$var1 * 2]
$ echo $var2
12
```

Для виконання операцій з плаваючою крапкою використовується вбудований в Bash калькулятор `bc`. Точність виконання операцій задається змінною `scale`.

```
$ bc -q
3.44 / 5
0
scale=4
3.44 / 5
.6880
Quit
$
```

Для використання калькулятора в сценарії використовується наступний синтаксис

```
variable=`echo "options; expression" | bc`
```

Приклад:

```
#!/bin/bash
var1=`echo " scale=4; 3.44 / 5" | bc`
echo The answer is $var1
```

5 Робота із стрічками

Заміна керуючих послідовностей `\a—Alert (bell), \b—Backspace, \cC—A control character C, \e—Escape character, \f—Form feed, \n—New line, \r—Carriage return,`

\t—Horizontal tab, \v—Vertical tab, \\—A literal backslash, \'—A single quote, \nnn—The ASCII octal, \xnnn—The ASCII hexadecimal відповідними символами (**$'…'**)

```
$ printf "%s\n" $'line 1\nline 2\nline 3'
line 1
line 2
line 3
```

Виведення довжини стрічки (**#**):

```
$ COMPANY="Nightlight Inc."
$ printf "%s\n" "${#COMPANY}"
15
```

Перевірка наявності змінної:

```
$ printf "Company is %s\n" \
"${COMPANY:?Error: Змінна Company не оголошена}"
```

Видобування підстрічки із заданої позиції і заданої довжини (**:n:m**):

```
$ printf "%s\n" "${COMPANY:5}"
light Inc.
$ printf "%s\n" "${COMPANY:5:5}"
light
```

Вилучення підстрічки за шаблоном (%, **#**, %%, **##**). Шаблон ставиться за іменем змінної. У підстрічці, яка повертається шаблон вилучається. Один знак шаблону повертає найменш можливу підстрічку, а два знаки – найбільш можливу підстрічку. Для шаблонів #, ## підстрічка повертається правіше від них, а для шаблонів %, %% – лівіше від них.

```
$ COMPANY="Nightlight Inc."
$ printf "%s\n" "${COMPANY#Ni*}"
ghtlight Inc.
$ printf "%s\n" "${COMPANY##Ni*}"
$ printf "%s\n" "${COMPANY##*t}"
Inc.
$ printf "%s\n" "${COMPANY#*t}"
light Inc.

$ printf "%s\n" "${COMPANY%t*}"
Nightligh
$ printf "%s\n" "${COMPANY%%t*}"
Nigh
```

Заміна підстрічки з використанням шаблону (**/**, **//**). Якщо після змінної слідує шаблон '/', то замінюється тільки перше співпадіння, а якщо шаблон '//' – замінюються усі співпадіння.

```
$ COMPANY="Nightlight Inc."
$ printf "%s\n" "${COMPANY/Inc./Incorporated}"
Nightlight Incorporated
$ printf "Company name is %s" "${COMPANY//i/I}"
Company name is NIghtlIght Inc.
```

6 Регулярні вирази

Регулярні вирази (РВ) є набором символів або метасимволів які задають шаблон для пошуку. Такі команди як grep, expr, sed, awk використовують РВ.

Є два типи регулярних виразів:

- на основі стандарту POSIX Basic Regelar Expression (BRE);
- на основі стандарту POSIX Extended Regelar Expression (BRE).

Стандарт BRE реалізований в утилітах і сценаріях Linux, а ERE – у мовах програмування високого рівня.

В регулярних виразах зарезервовані спеціальні символи (метасимволи)

. * [ ] ^ $ { } \ + ? | ( )

Якщо в шаблоні РВ потрібний один із спеціальних символів, то його потрібно екранувати зворотньою косою рискою «\».

РВ може містити:

• любу символьну стрічку;

• якір, вказує позицію у стрічці, яка має співпасти із РВ (^– на початку, $ – в кінці стрічки);

• модифікатори, які розширюють або звужують текст, в якому здійснюється пошук на співпадіння з РВ.

Найчастіше РВ використовують для пошуку і маніпуляцій із стрічками. РВ можуть описувати окремі символи або їх набори символів, стрічки або їх частини.

1. ***** – нуль або більше повторень стрічки або РВ, що знаходиться перед ним (`string*`, `РВ*`);

"113*" співпадіння з *11 + нуль або більше повторів 3*: *11*, *113*, *1133*,…

```
COMPANY=”Альбатрос”
if [[ $COMPANY = A* ]] ; then
printf ”Назва компанії починається з букви A\n”
fi
```

2. **.** – співпадіння з любим символом, крім `newline`;

"13." Співпадіння з *13 + один любий символ (включно з пропуском)*: *13*, *133*, *134*

3. **^, $** – якорі співпадіння на початку (”^XXX”) і в кінці (“XXX$”) стрічки.

4. **[...]** – один символ з набору для співпадіння ([xyz] – x, y або z, [0-9] – одна з цифр).

```
if [[ $COMPANY = [ABC]* ]] ; then
printf ”Назва компанії починається з символу A, B або C\n”
fi
```

**** – символ екранування спеціальних символів;

\\<...\\> – виділення границь слова \\<123\\> виділяє 123, а не 1234.

Розширені РВ

1. **?** – нуль або одне повторення попередньої стрічки або РВ;

```
COMPANY=”МИР”
if [[ $COMPANY = M?? ]] ; then
printf ”Назва компанії має 3 символи і починається з символу M\n”
fi
```

2. **+** – одне або більше повторень повторення попередньої стрічки або РВ;

```
# GNU версія sed і awk використовують "+",
# але його необхідно екранувати символом ”\”.
echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
```

3. **\{ … \}** – число повторень попереднього РВ.

"[0-9]\{5\}" – співпадіння п'яти чисел.

4. **(…)** – об’єднання групи РВ.

5. **|** - задання набору альтернативних символів.

```
egrep 're(a|e)d' misc.txt
```

6. Регулярні вирази POSIX

[:alpha:] співпадіння з символами букв. Еквівалентно до **A-Za-z**.

[:blank:] співпадіння з символами пропуску і табуляції.

[:cntrl:] співпадіння з символами керуючими символами.

[:digit:] співпадіння з символами цифр. Еквівалентно до **0-9**. ·

[:graph:] (графічні друковані символи). Співпадіння з символами у діапазоні ASCII 33 - 126. Еквівалентно до **[:print:]**, крім символу пропуску.

[:lower:] співпадіння з символами малих букв. Еквівалентно до **a-z**.

[:print:] (друковані символи). Співпадіння з символами у діапазоні ASCII 32 - 126. Еквіваленто до **[:graph:]**, крім символу пропуску.

[:space:] співпадіння з символами пропуску і горизонтальна табуляція.

[:upper:] співпадіння з великими буквами. Еквівалентно до **A-Z**.

[:xdigit:] співпадіння з символами шістнадцяткових цифр. Еквівалентно до **0-9A-Fa-f**.

Приклади використання інструментів `sed` і `awk` для пошуку точного входження послідовності символів у текст:

```
$ echo ”Привіт всім” | sed –n '/всім/p'
$ echo ”Привіт всім” | awk '/всім/{print $0}'
```

Пошук входження символів у заданому файлі з використанням діапазонів символів:

```
$ echo '/var[0-9][0-9]/{print $0}' my.sh
var01
var25
```

7 Розширення імен файлів у Bash

Сам Bash не розпізнає РВ, але здійснює розширення імен файлів – цей процес відомий як *globbing*. У цьому процесі розпізнаються і розширюються тільки деякі спеціальні символи, що є суттєвим обмеженням:

***** – співпадіння з любими стрічками, в тому числі і порожніми (для імен файлів, крім тих, що починаються з символу ”**.**”).

```
$ echo *
a.1 b.1 c.1 t2.sh test1.txt
```

```
$ echo t*
t2.sh test1.txt
```

? – співпадіння з любим одиночним символом.

```
$ echo t?.sh
t2.sh
```

[…] – співпадіння з одним із заданого набору символів.

```
$ ls -l [a-c]*
```

^, ! – співпадіння з символами, крім заданих.

```
$ ls -l [^a-c]
$ ls -l [!a-c]
```

При використанні команди **shopt -s extglob** Bash виконує додаткові розширення імен файлів:

- ?(*pattern-list*)– пошук нуль або одного співпадіння з шаблоном
- *(*pattern-list*)– пошук нуль або більше співпадінь з шаблоном
- +(*pattern-list*)– пошук одного або більше співпадінь з шаблоном
- @(*pattern-list*)– пошук тільки одного співпадіння з шаблоном
- !(*pattern-list*)– пошук всього за винятком шаблону

Список шаблонів *pattern-list* у круглих дужках можна розділити символом ”|”.

```
if [[ $COMPANY = +(A)*(Ltd|Corp|Inc) ]] ; then
printf ”Назва компанії починається з одного або більше символів A і закінчується одним із слів Ltd, Corp, Inc.\n”
fi
```

```
$ ls *+(.c|.h)
actions.c coledit.c config.c
```

```
$ wc -l *+(.c|.h)
96 actions.c
201 coledit.c
```

```
24 config.c
321 total
```

{…} – дозволяють отримувати перестановки шаблонів

```
$ echo {0,1}{2,3}
02 03 12 13
$ echo {0,1}{1..3}
01 02 03 11 12 13
$ echo {/user1/,/user2/}.profile
/user1/.profile /user2/profile
```

8 Інструкції для організації циклів

Інструкція **for** дозволяє організувати цикл, який перебирає серію значень

```
for var in list
do
  commands
done
```

Виведення вмісту поточного каталогу:

```
$ for var in `ls –l` ; do echo $var; done
```

Цикл для перебору значень списку:

```
#!/bin/bash
echo -n "Друк в рядок десяти крапок"
for i in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "."
done
```

Добавлення розширення `.txt` для всіх файлів поточного каталогу:

```
#!/bin/bash
for file in *
do
  echo "Добавлення розширення .txt для файлів $file ..."
  mv $file $file.txt
  sleep 1
done
```

Порядковий друк текстового файлу:

```
#!/bin/bash
read –p ”Введіть ім’я файлу: ” my_file
for var in $(cat $my_file)
do
  echo " $var"
done
```

Зміна змінної середовища $IFS

```
#!/bin/bash
file=”/etc/passwd”
IFS=$’\n’
for var in $(cat $file)
do
  echo " $var"
done
```

Цикли у стилі мови програмування С:

```
for (( i=1; i<=3; i++ ))
```

```
do
  echo "Цикл у стилі C: $i"
done
```

Використання у циклі різних змінних і перенаправлення виведення у файл:

```
for (( a=1, b=10; a <= 10; a++, b-- ))
do
  echo "$a - $b"
done > ”my_file.txt”
```

Інструкція **`while умова`** використовується для організації циклів з перевіркою умови. *Цикл виконується поки `умова` істинна*:

```
#!/bin/bash
while true
do
  echo "Натисніть CTRL-C для виходу."
done
```

`true` – команда, яка запускає тіло циклу на повторення. Використання `true` вважається повільним, так як сценарій має її запускати в кожній ітерації. У альтернативному варіанті використовується вбудована команда Bash “:” (`type :` - is a shell builtin)

```
#!/bin/bash
while :
do
  echo " Натисніть CTRL-C для виходу."
done
```

Приклад організації циклу з 10 ітерацій з використанням змінних:

```
#!/bin/bash
x=0;
while [ ”$x” -le 10 ]
do
  echo "Поточне значення x: $x"
  # Збільшуємо значення x:
  # expr - команда для виконання арифметичних виразів
  # x=$(expr $x + 1)
  x=$[ $x + 1]
  sleep 1
done
```

Інструкція **`until умова`** використовується для організації циклів з перевіркою умови. *Цикл виконується поки `умова` не істинна*. Так в прикладі цикл зупиниться, коли величина `x` досягне значення 10.

```
#!/bin/bash
x=0
until [ ”$x” -ge 10 ]
do
   echo ”Поточне значення x дорівнює $x”
   # x=$(expr $x + 1)
   x=$[ $x + 1]
   sleep 1
done
```

Для виходу із циклу використовується інструкція **`break [ n ]`**, де `n` – номер вкладеності зовнішнього циклу.

Для пропущення поточної ітерації і продовження циклу використовується інструкція **`continue [ n ]`**, де `n` – номер вкладеності зовнішнього циклу.

Перенаправлення даних із циклів:

- перенаправлення у файл `do … done > out.txt`
- перенаправлення в іншу команду `do … done | sort`

9 Службові символи, які використовуються у сценаріях Bash

– початок коментарію
; – розділювач команд
;; – розділювач в команді *case*
. – оператор крапка
“ – екранування
‘ – строге екранування
, – кома
\ – екранування окремого спеціального символу
/ – префікс шляху
` – Підставлення команди
: – порожня команда
! – логічне заперечення
* – груповий шаблон
? – перевірка умови
$ – підставлення змінної
${} – підставлення змінної
$***, **$@ – аргументи командного рядка
$? – код завершення операції
$$ – PID процесу сценарію
$! – PID останнього запущеного у фоні процесу
( ) – група команд
{xxx,yyy,zzz,...} – фігурні дужку
{ } – блок коду (вкладений блок)
{} **;** – Шлях до файлу і його ім'я
[ ] – синонім команди *test*
[[ ]] – подвійні квадратні дужки
(( )) – подвійні круглі
>, &>, >&, >>, < – перенаправлення введення/виведення
<< – перенаправлення на вбудований документ
<, > – посимвольне ASCII-порівняння
\<, \> – границя слова
| – конвеєр
>| – примусове перенаправлення
|| – логічне АБО
& – виконати процес у фоновому режимі
&& – логічне І
- – дефіс
= – символ дорівнює
+ – плюс
% – модуль
~ – домашній каталог (тильда)
~+ – поточний робочий каталог
~- – попередній робочий каталог
^ – початок стрічки
  – символ пропуску

 # – початок коментаря.

Все що слідує за цим символом, є коментарем, за винятком тільки комбінації **#!**, яка розміщена у першому рядку. Любі команди, що йдуть за цим символом в одній стрічці,

будуть вважатися коментарями і виконуватися не будуть.

Якщо даний символ екранований або взятий в одинарні або подвійні лапки, він буде сприйматися як звичайний символ. Крім того він може використовуватися в операціях підставлення параметрів і в константних числових виразах.

```
echo "Тут символ # не є початком коментаря"
echo 'Тут символ # також не є початком коментаря'
echo Тут екранований символ \# також не є початком коментаря
echo А тут # означає коментар
echo ${PATH#*:} # Підставлення параметрів, символ # не є початком коментаря.
echo $(( 2#101011 )) # Основа системи числення, символ # не є початком
коментаря
```

Екранується символ #, символом \.

Крім вищеописаних ситуацій, символ # не інтерпретується як початок коментаря в операціях пошуку за шаблоном.

; – розділювач команд (крапка з комою).

Дозволяє записувати більше однієї команди в рядку.

```
echo hello; echo there
```

У деяких ситуаціях, даний службовий символ необхідно екранувати, як і знак початку коментаря.

;; – **розділювач в операторі *case* (подвійна крапка з комою)**

```
case "$variable" in
abc) echo "$variable = abc" ;;
xyz) echo "$variable = xyz" ;;
esac
```

. – крапка, аналог вбудованої в оболонку bash, команди source.

При використанні всередині сценарію, дозволяє підвантажувати зовнішній файл, наприклад з даними або функціями.

Наприклад, є файл *test.sh* який містить наступний сценарій:

```
#!/bin/bash
. var.file # завантажуємо змінні із зовнішнього файлу
echo $var  # виводимо значення змінної
```

Файл *var.file* містить всього один рядок із значенням:

```
var="Hello world !"
```

Запуск сценарію *test.sh* на виконання

```
>./test.sh
Hello world !
```

“Крапка” також може бути частиною імені файла. Якщо ім'я файла починається з крапки, як правило, такий файл буде скритим для перегляду командою *ls* (залежить від оболонки).

Якщо справа йде про каталоги, одна крапка означає поточний каталог, дві крапки каталог рівнем вище, тобто батьківський.

Команду *крапка*, зручно використовувати при копіюванні або переміщенні об'єктів файлової системи. Наприклад:

```
>cp /path/to/dir/* . # Копіювати всі файли з каталогу /path/to/dir/ у
поточний каталог
```

При операціях пошуку за шаблоном і в регулярних виразах, символ *крапка* означає любий одиночний символ.

“ – **подвійні лапки**.

У стрічці, взятій у подвійні лапки, не інтерпретується (екранується) більшість службових символів.

‘ – одинарні лапки.

Більш строгий варіант екранування. У стрічці, взятій в одинарні лапки, не будуть інтерпретуватися любі службові символи.

, – кома.

Команда *кома*, використовується для обчислення декількох арифметических виразів. Недивлячись на те, що обчислені будуть всі вирази, результат буде виведений тільки з останньої операції. Наприклад, такий сценарій:

```
#!/bin/bash
let ”result=((a=5+3, b=7-1, c=15-4))”
echo $a
echo $b
echo $c
echo $result
```

виведе наступні результати:

```
>./test.sh
8  # результат першої операції,  присвоєний змінній $a
6  # результат другої операції,  присвоєний змінній $b
11 # результат третьої операції, присвоєний змінній $c
11 # результат останньої операції дорівнює третій операції, оскільки вона остання в послідовності виразів, присвоєний змінній $result
```

\ – зворотня похила (Escape, зворотній слеш).

Використовується для екранування окремих службових символів у стрічці. За ефектом дії даний символ аналогічний одинарним лапкам:

```
echo '$' # надрукує символ $
echo \$ # також надрукує символ $
```

/ – похила (слеш).

Використовується як розділювач в шляхах каталогів і файлів, в ОС Unix, Linux. При використанні в арифметичних операціях, означає ділення.

` – підставлення команд (лівонахилені лапки).

Можуть використовуватися для присвоєння змінній, результатів виконання системних команди. Сценарій:

```
#!/bin/bash
result= `hostname` # результат команди hostname присвоюється змінній
echo $result
```

виведе

```
>./test.sh
Linux.suse     # результат виконання команди hostname
```

: – порожня операція (двокрапка).

Є аналогом операції *“NOP”* (немає операції). Дану команду можна вважати еквівалентом вбудованої команди *true*. Як і *true*, команда *двокрапка*, завжди повертає **0** (нуль), тобто значення істина *(true)*.

Практично усі команди в операційних системах *unix*, *linux* повертають "**0**" у разі успішного завершення роботи.

Можливі наступні варіанти використання.

Нескінченний цикл:

```
while :
do
```

```
   echo "test"
done
```

```
# теж саме, але з використанням true
# while true
# do
#   echo "test"
# done
```

Символ заповнювач в умовному операторові *if/then*.

```
a=1
b=2
if [ $a = $b ]
then : # Не виконувати ніяких дій
else
   echo "test" # умова повертає false( $a не рівне $b )
fi
```

Символ заповнювач в операціях, що припускають наявність двох операндів.

```
: ${username='whoami'} # без символу двокрапки з пропуском буде видано
повідомлення про помилку
```

Символ заповнювач в конструкціях вкладений документ.
В операціях з підставленням параметрів.

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
```

В операціях заміни підрядка з підставленням змінних.
В поєднанні з операторами перенаправлення виведення. При використанні з оператором >, обнуляє вміст файлу, якщо файлу не існує, він створюється:

```
: > data.file # цією командою файл data.file буде очищений до нуля
```

Того ж можна добитися за допомогою команди `cat /dev/null > data.file`, тільки в цьому випадку створюється новий процес.
У поєднанні з оператором >> перенаправлення з добавленням у кінець файлу і зміною часу останнього доступу.

```
: >> data.file
```

При заданні імені не існуючого файлу, він буде створений, аналогічно вищеописаному варіанту або дії команди *touch*.
Два попередні варіанти незастосовні до символічних посилань, конвеєрів і інших спеціальних файлів.
Також символ *двокрапка* використовується як роздільник полів у файлі */etc/passwd* і змінній оточення *$PATH*.

```
>echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:/root
/bin
```

! – логічне заперечення в операторах умов.

Символ *знак оклику* використовується для інвертування коду повернення операції до якої він застосовується. Так-же застосовується для логічного заперечення в операціях порівняння :

```
if [ $a = $b ] # істина, якщо $a дорівнює $b
if [ $a != $b ] # ця умова істинно якщо $a НЕ ДОРІВНЮЄ $b
```

Знак оклику є зарезервованим словом оболонки *bash*.
В деяких випадках може бути використаний для непрямого звернення до змінних. При використанні з командного рядка оболонки, запускає механізм роботи з історією команд, з сценаріїв ця можливість не доступна.

*** – символ групового шаблону(зірочка).**

Використовується як шаблон для підставлення в імені файлу. У разі використання поодинокого символу шаблону, означає співпадіння з будь-яким ім’ям файлу.

```
>echo *
file1.sh file2.sh file3.sh arch1.tar arch2.tar
```

У наступному сценарії будуть виведені файли з будь-яким ім'ям, але тільки з розширенням *tar*:

```
>echo *.tar
arch1.tar arch2.tar
```

У регулярному виразі знак * означає 0 або більше повторів попереднього символу або групи символів (у круглих дужках).
В арифметичних операціях знак * – означає множення, а у варіанті ** – піднесення до ступеня.

? – перевірка умови.

В деяких випадках використовується для перевірки виконання умови.
При використанні в конструкціях з подвійними дужками задає тернарний оператор мови C.

```
echo $(( t = a<6 ? 10 : 20 )) # Тернарний оператор
if (( a < 6 )); then t=10; else t=20; fi; echo $t # традиційний варіант
```

При використанні у виразах з підставленням параметрів, перевіряє чи має значення змінна.

```
: ${HOSTNAME?} ${USER?} ${MAIL?} # якщо одна або декілька змінних не визначені, буде виведено повідомлення про помилку
```

Використовується як символ шаблон. При підставленні в ім’я файлу означає *один символ*, в регулярних виразах, означає повтори ввід 0 до 1 попереднього символу.

$ – підставлення змінної

```
a=5
b=10
echo $a # виведе 5
echo $b # виведе 10
```

Розміщення символу **$** перед ім’ям змінної, означає що буде отримане значення цієї змінної.
У регулярних виразах означає *кінець рядка*.

${} – підставлення змінної.

Виконує ту ж дію, що і попередній варіант, але в деяких випадках при неоднозначності інтерпретації, працюватиме тільки цей варіант. Крім того може бути використаний для зчеплення стрічкових змінних в один рядок.

```
id=${USER}-on-${HOSTNAME}
echo "$id"
```

виведе

```
>./test.sh
ki21_15-on-linux
```

$*, $@ – аргументи командного рядка.

Змінна **$*** містить усі параметри командного рядка як одну стрічку.
Змінна **$@** теж містить усі параметри командного рядка, але кожний параметр як окрему стрічку.

$? – Код завершення операції.

Ця змінна отримує код, повернутий останньою виконаною командою, функцією або сценарієм.

$$ – PID процесу

Змінна містить ідентифікатор процесу поточного сценарію *process id*.

( ) – Група команд

Команди розміщені в дужках, виконуються у дочірньому процесі (*subshell*). При цьому змінні, визначені у дочірньому процесі, не видимі у батьківському.

```
var=12345
( var=54321; ) # змінна визначається в дочірньому процесі
echo "var = $var"
```

виведе

```
>./test.sh
a = 12345
```

Інший варіант використання конструкції з одинарними дужками, ініціалізація масивів.

```
array= (element1 element2 element3)
```

{xxx, yyy, zzz...} – фігурні дужки

Інтерпретується як список можливих варіантів. Наприклад, наступна команда сценарію, шукає стрічку *test* у файлах *data.1*, *data.2*, *data.3*.

```
grep test data.{1,2,3*}
```

і виводить знайдені варіанти

```
>./test.sh
data.1: test
data.3: test
```

Щоб використати у фігурних дужках пропуски потрібно взяти їх в одинарні або подвійні лапки, або екранувати їх за допомогою символу **}.**_

```
echo {test1, test2}\ :{\ A," B",' C'}
виведе
>./test.sh
test1 : A test1 : B test1 : C test2 : A test2 : B test2 : C
```

{} – Блок коду (вкладений блок)

По суті така конструкція створює анонімну функцію. У відмінності від традиційно створюваних функцій, змінні *вкладеного блоку*, доступні усьому сценарію.

```
a="test"
{ a=1111; }
echo "a = $a"
```

буде виведено:

```
>./test.sh
a = 1111
```

Конструкція у фігурних дужках, може створювати перенаправлення введення/виведення. Перенаправлення введення/виведення з вкладеного блоку:

```
#!/bin/bash
file=/etc/fstab
{
  read line1
  read line2
} < $file # читуємо рядки з файлу fstab
echo "$line1"
echo "$line2"
```

```
exit 0
```

в результаті виконання будуть виведені два перші рядки файлу `/etc/fstab`

```
>./test.sh
# Device                Mountpoint      FStype  Options         Dump    Pass#
/dev/da0s1b             none            swap    sw              0       0
```

Зберегти результат роботи вкладеного блоку у файл

```
#!/bin/bash
# rpm - check.sh
# Цей сценарій отримує опис rpm- пакету, список файлів, і перевіряє
можливість установки.
# Результат роботи зберігається в окремому файлі.
SUCCESS=0
E_NOARGS=65
if [ -z "$1" ]
then
  echo "Порядок використання : ’basename $0’ rpm - file"
  exit $E_NOARGS
fi
{
echo
echo "Опис архіву :"
rpm -qpi $1 # Отримуємо опис rpm пакету
echo
echo "Список файлів :"
rpm -qpl $1 # Отримуємо список файлів пакету
echo
rpm -i --test $1 # Перевіряємо, чи можна встановити цей пакет
if [ "$"? -eq $SUCCESS ]
then
  echo "$1 встановлення можливе"
else
  echo "$1 встановлення не можливе"
fi
echo
} > "$1.test" # Перенаправлення результатів в зовнішній файл.
exit 0
```

На відміну від конструкцій із звичайними одинарними дужками, що виконуються в дочірньому процесі, вкладені блоки у фігурних дужках, виконуються у рамках того ж процесу, що і сам сценарій.

{} \; – Шлях до файлу і його ім'я

Найчастіше застосовується з командою *find* і опцією *–exec*.
Зверніть увагу що символ *крапка з комою*, що завершує опцію *-exec*, команди *find*, має бути екранований, щоб уникнути його інтерпретації.

Взнати шлях, ім’я і розширення файлу:

```
full=”/aaa/bbb/c.sh”
path=”${full%/*}
name=”{full##*/}”
exten=”{full##*.}”
```

[ ] – test

У конструкції з квадратними дужками, перевіряється істинність включеного в нього виразу.

```
var=1
if [ $var = 1 ]
then
  echo "var= $var"
```

```
else :
fi
```

При роботі з масивами, в *квадратних дужках* вказується індекс елементу до якого треба звернутися.

```
array= (a b c)   # створюємо масив
echo ${array[1]} # виводимо другий елемент масиву
```

Нумерація елементів масиву починається з 0 (нуля), тому в елементі з індексом **1**, знаходиться символ **b** .
При використанні в регулярних виразах, в квадратних дужках, пишуться так звані класи символів або діапазони:
[xyz] – відповідає будь-якому з цих трьох символів.
[0-9] – відповідає будь-якому числовому символу від 0 до 9.
[a-zA-Z] – відповідає будь-якій букві англійського алфавіту.

[[ ]] – Подвійні квадратні дужки

Це розширений варіант конструкції з одинарними *квадратними дужками*. В *подвійних квадратних дужках* також перевіряється істинність поміщеного в цю конструкцію виразу, але цей варіант прийнятніший, оскільки дозволяє уникнути деяких логічних помилок. Наприклад в конструкції з *подвійними квадратними дужками*, можна використати оператори **&&**, **||**,**<** і **>**.
При використанні з умовним оператором *if*, наявність квадратних дужок, як одинарних так і подвійних, не обов'язкова.

(( )) – Подвійні круглі дужки

Між *подвійними круглими дужками*, обчислюється цілочисельне арифметичний вираз. Крім того подвійні дужки дозволяють працювати із змінними в стилі мови С:

```
#!/bin/bash
echo (( a = 23 )) # присвоєння значення змінній
echo "a( )  = $a"
(( a++ )) # після-інкремент значення змінної $a, в стилі мови C
echo "a( a++)  = $a"
(( a-- )) # після-декремент значення змінної $a, в стилі мови C
echo "a( a--)  = $a"
(( ++a )) # перед-інкремент значення змінної $a, в стилі мови C
echo "a( ++a)  = $a"
(( --a )) # перед-декремент значення змінної $a, в стилі мови C
echo "a( --a)  = $a"
```

також можна використати тернарні оператори, згадані раніше :

```
(( t = a<6?10: 20 ))
```

>, &>, >&, >>, < – перенаправлення введення/виведення

У будь-якій Unix, Linux системі, за замовчування відкриті три файли *stdin* (стандартний потік введення – клавіатура), *stdout* (стандартний потік виведення – екран) і stderr (стандартний потік виведення помилок), дескриптори **0, 1** і **2**, відповідно.
Оператори перенаправлення дозволяють передати виведення з файлу, сценарію, команди або блоку команд на введення іншого файлу, сценарію, команди. Наприклад:

```
>./test.sh > outfile  # перенаправлення stdout у файл outfile
>command &> outfile # перенаправлення  stdout і stderr команди у файл outfile
>command >&2 # перенаправлення stdout команди у потік stderr
>./test.sh >> outfile # перенаправлення stdout у файл, в режимі додавання у кінець файлу
```

Операція підставлення процесу, передає виведення одного процесу на введення іншого.

```
 (command)>
<(command)
```

між символами <, > і круглою дужкою не має бути пропуску.

Крім того символи < і > використовуються в операціях порівняння символів і цілих чисел.

<< – перенаправлення на вбудований документ

Вбудований документ, спеціальний вид перенаправлення, який дозволяє передати інтерактивній команді або програмі, список команд. Наприклад, сценарій з виведенням багаторядкового тексту за допомогою програми *cat*, може виглядати так:

```
#!/bin/bash
cat <<End-of-message
line one
line two
line three
End-of-message
exit 0
```

на виході отримаємо наступне:

```
>./test.sh
line one
line two
line three
```

Символ-обмежувач, в нашому прикладі *End-of-message*, не повинен повторюватися в тілі самого *вбудованого документу*.

<, > – посимвольне ASCII- порівняння

Порівнюються ASCII коди символів.

```
if [[ "$a" < "$b" ]]
if [ "$a" \< "$b" ]
if [[ "$a" > "$b" ]]
if [ "$a" \> "$b" ]
```

Зверніть увагу, при використанні операцій порівняння в одинарних квадратних дужках, символи *більше* і *менше, необхідно екранувати зворотною похилою.*

\<, \> – границя слова

Ці символи використовуються для позначення границь слова в регулярних виразах, наприклад в операціях пошуку.

```
grep '\<ec' file  # знайти усі слова, що починаються на ec
grep 'ho\>' file  # знайти усі слова, що закінчуються на ec
```

| – конвеєр

Конвеєр (канал, pipe), це класичний спосіб міжпроцесної взаємодії, *stdout* одного процесу, передається на *stdin* іншого. Часто використовується для зв'язування декількох команд між собою. Одна команда, передає результат оброблення даних через конвеєр на введення іншої команди. Наприклад, виведення команди *ps* (process status – список процесів), передається на введення команді *grep* (пошук за шаблоном), яка у свою чергу, зробивши вибірку, виводить результат в *stdout*.

```
>ps aux | grep sshd
root    1274  0.0  0.2 25108  4460  ??  Is   Mon03PM   0 : 00.01
/usr/sbin/sshd
root    1349  0.0  0.2 37040  5164  ??  Ss   Mon03PM   0 : 06.84 sshd:
root@pts/0(sshd)
root   22977  0.0  0.1  8060  1396   1  RL+   9 : 18PM   0: 00.00 grep sshd
```

Інший приклад, команда *cat* виводить вміст файлу на вхід команди *wc*, яка підразовує кількість рядків і виводить результат.

```
>cat test.sh | wc -l
9
```

У конвеєри можна об'єднувати як команди, так і сценарії. Наприклад, перенаправимо виведення команди *ls* на вхід сценарію, який перетворює символи у верхній регістр:

```
#!/bin/bash
tr 'a-z' 'A-Z'
exit 0
```

```
>ls - l | ./test.sh # робимо перенаправлення
```

```
TOTAL 6688
-RWXR--R--   1 ROOT  WHEEL       28 JUL 25 18 : 34 1.PHP
-RWXR--R--   1 ROOT  WHEEL       75 APR  4 09 : 57 1.PL
-RWXR--R--   1 ROOT  WHEEL       60 APR  4 16 : 47 1.SH
```

Усі символи, виведення команди *ls* перетворені у верхній регістр.
Stdout кожного процесу в конвеєрі, повинен читатися в *stdin* іншого процесу, інакше конвеєр обірветься:

```
cat file | ls -l | sort # тут виведення утримуваного файлу командою cat піде
в нікуди, в результаті на виході отримаємо не те, що очікуємо
```

Конвеєр виконується в окремому процесі, тому не може отримати доступ до змінних сценарію.
Якщо якась з команд конвеєра завершується аварійно, увесь конвеєр аварійно завершує роботу.

>| – Примусове перенаправлення

Перенаправлення відбувається навіть якщо *bash*, запущено з ключем **– C** (*noclobber)*.

|| – Логічне АБО

У операціях порівняння, ||, повертає значення *true*, якщо хоча б одно умова повертає *true*:

```
#!/bin/bash
a=1
b=2
if [ $a -eq 10 ] || [ $b -eq 2 ]
then
  echo "true"
else
  echo "false"
fi
```

цей невеликий сценарій виведе *true*, оскільки друга умова поверне *true*, то і уся перевірка поверне *true*. Наведену вище умову можна записати в наступній формі:

```
if [[ $a -eq 10 || $b -eq 2 ]]; then echo yes; fi
```

або аналог

```
if [ $a -eq 10 -o $b -eq 2 ]; then echo yes; fi
```

Оператор ||, не може використовуватися в *одинарних квадратних дужках*.

& – Виконати процес у фоновому режимі

Команда, за якою стоїть символ **&** *(амперсанд)*, виконуватиметься у фоновому режимі.
У сценаріях, у фоновому режимі можна виконувати не лише команди, але і цикли, наприклад:

```
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$i "
done &
echo
```

```
for i in 11 12 13 14 15 16 17 18 19 20
do
  echo -n "$i "
done
echo
exit 0
```

Зверніть увагу, команда, запущена з сценарію у фоновому режимі, може перекрити виведення фонової команди. Наприклад:

```
#!/bin/bash
ls -l & # команда ls запускається у фоні
echo "Done"
```

виведе такий результат:

```
>./test.sh
Done
```

як видно, результат виконання команди *ls*, відсутній.

Щоб виправити поведінку сценарію, досить використати оператор *wait*, який призупиняє виконання сценарію, до тих пір, поки не будуть завершені усі фонові завдання. Приклад:

```
#!/bin/bash
ls -l &
echo "Done"
wait
```

тепер буде очікуване виведення:

```
>./test.sh
Done.
total 6688
-rwxr--r--   1 root  wheel       28 Jul 25 18 : 34 1.php
-rwxr--r--   1 root  wheel       75 Apr  4 09 : 57 1.pl
-rwxr--r--   1 root  wheel       60 Apr  4 16 : 47 1.sh
```

&& – логічне І

У умовних операціях, **&&**, повертає *true* тільки у тому випадку, якщо ОБИДВА операнди мають значення *true*.
Формат використання аналогічний операторові **||** (*логічне АБО*).

```
if [ $a -eq 1 ] && [ $b -eq 2 ] ; then echo yes; fi
```

або

```
if [[ $a -eq 1 && $b -eq 2 ]] ; then echo yes; fi
```

або

```
if [ $a -eq 1 -a $b -eq 2 ]
```

- – дефіс

Часто дефіс передує опціям команд.

```
ls -l
```

Для багатьох команд наявність дефіса перед ключами опція не обов'язково, наприклад у команди *ps*, проте бувають ситуації коли його використання потрібне, щоб опції інтерпретувалися саме як опції а не як їх значення.

Перенаправлення з/в *stdin* або *stdout*
Приклад переміщення дерева каталогів і файлів за допомогою архіватора *tar* :

```
(cd ./ss && tar cf - . ) | (cd ./test && tar xpvf -)
```

Що робить цей сценарій:
• *cd ./ss* – перейти в каталог у поточному каталозі, вміст якого переміщатиметься;
• *&&* – усі подальші команди будуть виконані тільки після виконання переходу в початковий каталог ;
• *tar cf* - . – ключ **c** вказує архіватору *tar* створити новий архів, ключ **f** (*file*) з подальшим **-** задає файл архіву – *stdout*, в архів поміщається увесь вміст поточного каталогу **(.)**;
• / (...) – конвеєр у дочірній процес;
• *cd ./test* – перейти в каталог призначення ;
• *&&* – як і вище;
• *tar xpvf* - – розпакувати *tar* архів (ключ **x**), зберегти власників і права доступу до файлів (ключ **p** , виводячи детальні дані під час виконання (ключ **v**), файл архіву stdin (ключ **f** з подальшим -).
Ще один приклад:

```
>echo "hello" | cat -
hello
```

У такому контексті **-** (дефіс), швидше не окремий оператор *bash*, а опція, яка розпізнається Unix утилітами (*tar*, *cat* і так далі), що виводять результати в *stdout*.
Якщо передбачається ім'я файлу, **-**, перенаправляє виведення в *stdout* або приймає введення з *stdin*.

При запуску програми *file* без параметрів, буде видано повідомлення про помилку:

```
>file
Usage: file [- bcikLhnNrsvz0] [- e test] [- f namefile] [- F separator] [- m
magicfiles] file...
```

якщо передати як параметр **-** (дефіс), *file* чекатиме призначене для користувача введення:

```
>file -
aaaaa
/dev/stdin: ASCII text

>file -
#!/bin/bash
/dev/stdin: POSIX shell script text executable
```

як видно, програма *file* проаналізувала стандартний потік введення *stdin* і визначила тип його вмісту.
Передача *stdout* по конвеєру на вхід інших команд, дозволяє виконувати різні трюки, наприклад, вставляти рядок на початок файлу :

```
#!/bin/bash
file="./text.txt"
title="this is first line"
echo $title | cat - $file > $file.new
```

на виході отримуємо файл *text.txt.new*, що містить усі попередні дані, плюс перший рядок "*this is first line"*.

Тепер повніший приклад використання архіватора *tar* з символом **-** для архівації файлів, які змінилися за останню добу:

```
#!/bin/bash
BACKUPFILE=backup
archive=${1 :-$BACKUPFILE} # якщо ім'я для архіву не задане в командному
рядку, буде призначено ім'я за умовчанням backup.tar.gz
tar cvf - 'find . - mtime - 1 - type f - print0 | xargs - 0 tar rvf
"$archive.tar"'
gzip $archive.tar
echo "done"
```

```
exit 0
```

Оператор перенаправлення **-** може конфліктувати з іменами файлів, які розпочинаються з дефіса, *- filename*, щоб цього уникнути, в сценаріях необхідно перевіряти імена файлів і задавати їм префікс шляху, наприклад: *./ - filename* або *$PWD/-filename*.

Крім того не забувайте про значення змінних, простий приклад:

```
var="-n"
echo $var
```

цей сценарій нічого не виведе, оскільки буде отримана команда *echo -n*, тобто значення змінної, *"-n"*, буде інтерпретоване як опція команди *echo*, яка просто подавлює виведення символів.
Команда *cd* з дефісом (*cd -*) виконує перехід у попередній робочий каталог.

У арифметичних виразах дефіс означає операцію віднімання.

= – символ дорівнює
Залежно від контексту застосування, виступає в ролі операції присвоєння значень змінним:

```
a=100
b="test"
echo $a # виведе 100
echo $b # виведе test
```

або як знак рівності в операціях порівняння.

+ – плюс
Залежно від контексту використання, оператор додавання в арифметичних операціях або квантифікатор у регулярному виразі, який означає один або більше повторів попереднього символу.

% – модуль
Залежно від контексту використання, залишок від ділення в арифметичних операціях або виступає шаблоном.
У шаблонах, вилучає із змінної більшу або меншу підстрічку, співпадаючу з шаблоном. Пошук ведеться з кінця стрічки.

~ – Домашній каталог (тильда)
Відповідає змінній оточення *$HOME*, що містить шлях до домашнього каталогу поточного користувача.

```
>echo ~
/root

>ls -l ~  # лістинг домашнього каталогу поточного користувача
total 47030
-rw-------  1 root        wheel            120 Aug 28 23 : 56 .bash_history
drwxr - xr - x  5 root        wheel            512 Jul 16 16 : 54 .cpan
-rw - r--r--  1 root        wheel            940 Jul  5 12 : 16 .cshrc
drwxr - xr - x  3 root        wheel            512 Jun 24 02 : 41 .gem
```

~+ – Поточний робочий каталог
Відповідає змінній оточення *$PWD*, що містить ім'я поточного робочого каталогу.

~- – Попередній робочий каталог
Відповідає змінній оточення *$OLDPWD*, що містить ім'я попереднього робочого каталогу.

^ – Початок рядка
При використанні в регулярних виразах, означає початок стрічки тексту.

Керуючий символ

Символ, який керує виведенням тексту або терміналом. Набирається на клавіатурі як комбінація *Ctrl+клавіша*.

Ctrl+C – перервати виконання процесу.
Ctrl+D – вихід з системної оболонки, аналог команди *exit* або *EOF* – кінець файлу, також є завершальним символом при введенні з *stdin*.
Ctrl+G – *BEL*, звуковий сигнал.
Ctrl+H – *Backspace*, вилучити попередній символ.
Ctrl+J – Повернення каретки.
Ctrl+L – Аналог команди *clear*, очищення вікна терміналу.
Ctrl+M – Переклад рядка.
Ctrl+U – Очистити рядок введення.
Ctrl+X – Призупинити виконання процесу.

Символ пропуску

Пропуском вважається сам символ пропуску, символ табуляції, переведення рядка, повернення каретки або комбінація з перерахованих символів. Використовується як роздільник команд і змінних.

Оскільки *bash* досить чутливий до пропусків, є строгі обмеження на їх використання в операціях присвоєння значень змінним. Наприклад:

```
var = 123 # помилка, змінна буде інтерпретована як команда з аргументами 123
var=123   # правильний варіант

let c = $a - $b # не правильний варіант
let c=$a -$b    # правильний
let "c = $a - $b" # допустимий варіант
if [ $a -le 5]    # не правильний варіант
if [ $a -le 5 ]   # правильний варіант
if [ "$a" -le 5 ] # правильний варіант
[[ $a -le 5 ]]    # правильний варіант
```

Порожні рядки ніяк не інтерпретуються, тому їх можна використовувати для візуального виділення рядків або блоків сценарію.

Змінна *$IFS* (за замовчуванням символ пропуску) містить роздільник полів, який використовується деякими програмами.

Висновки.

• Bash є не тільки командною оболонкою, але і мовою написання сценаріїв.
• Сценарії дозволяють використати всі можливості командної оболонки і автоматизувати задачі, які потребують для свого виконання введення багатьох команд.
• Мова Bash має достатній набір інструкцій для виконання команд за складною логікою.

Література.

[1] C. Newham, B. Rosenblat. Learning the bash shell. Third Edition. O’Reilly, 2005. – 333 p.

Запитання.

1. У яких випадках використовуються мови сценаріїв.
2. Перерахувати ключові слова мови Bash.
3. Які секції може містити сценарій Bash?
4. Як визначити чи команда належить оболонці Bash чи ОС Linux?
5. Які особливості оголошення і використання змінних Bash?
6. Призначення і особливості використання вбудованої команди Bash `test`?

7. Особливості синтаксису і використання інструкції `if` і `case` в Bash.
8. Інструкції для організації циклів в Bash.
9. Можливості і особливості виконання арифметичних виразів в мові Bash.
10. Робота із стрічками в мові Bash.
11. Регулярні вирази у мові Bash.

9. СТВОРЕННЯ І ВИКОРИСТАННЯ ФУНКЦІЙ У BASH

Мета. Вивчення особливостей створення, виклику і передачі параметрів функціям у сценаріях Bash.

Вступ. Використання функцій у Bash дозволяє писати більш складні сценарії. Функції можуть отримувати параметри із сценаріїв і повертати їм коди завершення. Для передачі параметрів функціям використовуються змінні оточення. Функції не можуть отримувати або повертати змінні масиви. Елементи масиву можна передавати або отримувати із функції як окремі значення. В сценаріях і функціях можуть використовуватися як глобальні, так і локальні змінні. Bash функції можуть викликати самі себе, тобто організовувати рекурсію. Функції, які використовуються в різних сценаріях, можна записати у бібліотечний файл.

План.
1 Створення і виклик функцій у Bash
2 Передача параметрів у функцію
3 Глобальні і локальні змінні
4 Масиви змінних і функції
5 Рекурсія
6 Бібліотечні файли функцій
7 Сценарій демон

1 Створення і виклик функцій у Bash

При написанні більш складних сценаріїв виникає необхідність у використанні функцій. У Bash функції це блок сценарію, який має ім'я. За цим іменем функція може викликатися з різних частин основного сценарію.

Є два формати синтаксису створення функції. Перший формат використовує ключове слова `function` і атрибут `name`, який задає ім'я функції:

```
function name {
  commands
}
```

де `commands` – одна або більше команд Bash. Кожна функція повинна мати унікальне ім'я.

Другий формат більш подібний до формату функцій у мовах програмування високого рівня:

```
name() {
  commands
}
```

Функції можна викликати без аргументів і з аргументами.

Для виклику функції у сценарії потрібно вказати її ім'я. Функції мають бути оголошені до їх виклику:

```
#!/bin/bash
# using a function in a script
function func1 {
  echo "This is an example of a function"
}
count=1
while [ $count -le 3 ]
do
  func1
  count=$[ $count + 1 ]
done
```

```
echo "This is the end of the loop"
func1
echo "Now this is the end of the script"

$ ./test1
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
This is an example of a function
Now this is the end of the script
```

За замовчуванням код завершення функції (exit status) є кодом, який повертає остання команда функції. Після завершення функції можна використати змінну `$?` для отримання коду повернення функції.

```
#!/bin/bash
# testing the exit status of a function
func1() {
  echo "trying to display a non-existent file"
  ls -l badfile
}
echo "testing the function:"
func1
echo "The exit status is: $?"

$ ./test4
testing the function:
trying to display a non-existent file
ls: badfile: No such file or directory
The exit status is: 1
```

Команда `return` дозволяє повернути із функції цілочисельний код із значенням від 0 до 255:

```
#!/bin/bash
# using the return command in a function
function dbl {
  read -p "Enter a value: " value
  echo "doubling the value"
  return $[ $value * 2 ]
}
dbl
echo "The new value is $?"
```

Результат виконання функції, а не цілочисельний код можна зберегти у змінній:

```
result=`db1`  # або result=$(db1)
```

Тоді значення функції `db1` буде присвоєне змінній `$result`, а її значення можна вивести командою `echo $result`. Приклад використання такого присвоєння:

```
#!/bin/bash
# using the echo to return a value
function dbl {
  read -p "Enter a value: " value
  echo $[ $value * 2 ]
}
result=`dbl`
echo "The new value is $result"

$ ./test
Enter a value: 200
The new value is 400

$ ./test
Enter a value: 1000
```

```
The new value is 2000
```

Функцію можна зробити доступною за межами сценарію, для чого потрібно її експортувати:

```
# test.sh
#!/bin/bash
prints(){
echo "Привіт всім"
}
declare -x -f prints

$ source test.sh
Привіт всім

$ prints
Привіт всім
```

2 Передача параметрів у функцію

Оболонка Bash розглядує функцію як невеликий сценарій. Це означає, що функції можна передати параметри як звичайному сценарію з командного рядка. Для визначення числа параметрів, які передаються у функцію використовується спеціальна змінна `$#`. Приклад передачі параметрів у функцію з командного рядка:

```
# test.sh
#!/bin/bash
# passing parameters to a function
function addem {
  if [ $# -eq 0 ] || [ $# -gt 2 ]
  then
    echo -1
  elif [ $# -eq 1 ]
  then
    echo $[ $1 + $1 ]
  else
    echo $[ $1 + $2 ]
  fi
}

$ echo -n "Додати 10 і 15: "
$ value=`addem 10 15`
$ echo $value
25


$ echo -n "Додати одне число: "
$ value=`addem 10`
$ echo $value
20

$ echo -n "Додоти без числа: "
$ value=`addem`
$ echo $value
-1

$ echo -n "Додати три числа: "
$ value=`addem 10 15 20`
$ echo $value
-1
```

Функція не може напряму працювати із змінними середовища Bash оболонки `$0, $1,...`, але їх можна передати у функцію як параметри. Для цього потрібно в сценарії вручну і явно передати параметри консольного рядка у функцію:

```
#!/bin/bash
# доступ до параметрів сценарію всередині функції
function func7 {
  echo $[ $1 * $2 ]
}
if [ $# -eq 2 ]
then
  value=`func7 $1 $2`
  echo "Результат $value"
else
  echo "Запуск: test.sh a b"
fi
```

```
$ ./test.sh
Запуск: test.sh a b
$ ./test7 10 15
Результат 150
```

3 Глобальні і локальні змінні

Область видимості змінної визначає її доступність у сценаріях і функціях. Змінні визначені у функціях можуть мати різну область видимості.

Функції використовують два типи змінних:

- глобальні;
- локальні.

Для явного оголошення змінних рекомендується використовувати команду `declare`. *Змінні оголошені в основному сценарії і функціях є глобальними*. Якщо глобальна змінна визначена в основному сценарії, то вона буде доступною і в середині функції. Якщо ж глобальна змінна визначена всередині функції, то вона буде доступною і в основному сценарії.

Для оголошення локальних змінних всередині функцій і присвоєння їм значень може використовуватися команда `local`, але вона не може встановлювати атрибути змінних. Тому рекомендується для оголошення локальних змінних замість команди `local` використовувати команду `declare`. Якщо змінна явно не оголошена (тобто є глобальною), то вона не знищується при завершенні функції, що може спричиняти неочікувані результати. Локальні змінні знищуються при завершенні роботи функції.

До до змінних визначених за межами функції можна звертатися із функції:

```
# test.sh
#!/bin/bash
# використання глобальної змінної для передачі значення
function fun1 {
  value=$[ $value * 2 ]
}
read -p "Введіть значення: " value  # глобальна змінна сценарію
fun1
echo "Нове значення: $value"
```

```
$ ./test.sh
Введіть значення: 450
Нове значення: 900
```

Люба змінна, яка використовується тільки всередині функції оголошується локальною:

```
local temp
```

Ключове слово `local` може використовуватися при присвоєнні значення функції:

```
local temp=$[ $value + 5 ]
```

Якщо в сценарії зустрічається змінна з іменем, що співпадає з іменем локальної змінної у функції, то сценарій розглядує їх як дві різні функції. Так в наступному сценарії використовуються дві різні змінні з однаковим іменем temp:

```
# test.sh
#!/bin/bash
# приклад локальної змінної
function func1 {
  local temp=$[ $value + 5 ]  # temp локальне
  result=$[ $temp * 2 ]
}

temp=4    # temp глобальне
value=6   # value глобальне

func1
echo "Результат $result"
if [ $temp -gt $value ]
then
  echo "temp=$temp є більшим $value"
else
  echo "temp=$temp є меншим $value"
fi

$ ./test.sh
Результат 22
temp=4 є меншим 6
```

Всі змінні Bash зберігаються як символьні стрічки. Кожна змінна має атрибути, які можна встановити командою declare, приклад оголошення цілочисельної змінної

```
>declare –i nomer=15
>printf ”%d\n” $i
>nomer=”Hello” # помилка: присвоєння цілочисельній змінній значення стрічки
```

Змінну можна оголосити константою declare -r (read-only).

Атрибути змінної або функції можна вивести командою

```
>declare –p my_variable
>declare –p my_function
```

Якщо змінні оголошена командою declare в основному сценарії, то вона є глобальною, а якщо всередині функції, то вона є локальною.

```
# Global Declarations
declare -rx SCRIPT=${0##*/} # SCRIPT є імя цього сценарію
declare -rx who=”/usr/bin/who” # команда who
```

Змінні оголошені командою declare існують до завершення функції або сценарію, або до їх руйнування командою unset. Для того щоб змінні були доступними за межами їх визначення іншим сценаріям, потрібно оголосити їх як експортовані (атрибут –x, export), наприклад declare -x CVSROOT=”/home/cvs/cvsroot”.

Для того щоб функція була доступною за межами її визначення іншим сценаріям, потрібно використати команду export

```
$export -f my_fun
```

4 Масиви і функції

Якщо спробувати передати як параметр у функцію масив, то функція сприйме тільки перший елемент масиву:

```
#test.sh
#!/bin/bash
# передача масиву у функцію
```

```
function testit {
echo "Параметри: $@"
thisarray=$1
echo "Отриманий масив ${thisarray[*]}"
}
myarray=(1 2 3 4 5)
echo "Оригінальний масив: ${myarray[*]}"
testit $myarray
```

```
$ ./test.sh
Оригінальний масив: 1 2 3 4 5
Параметри: 1
./test: thisarray[*]: bad array subscript
Отриманий масив
```

Тому, для передачі масиву у функції потрібно видобувати значення елементів масиву в окремі значення і передавати їх як параметри у функцію. Всередині функції із отриманих параметрів можна відновити масив.

Наступний сценарій використовує змінну `$myarray` для зберігання окремих значень масиву і передачі їх у функцію через командний рядок. Функція відновлює змінну масив з отриманих параметрів і використовує його для обчислення суми елементів:

```
# test.sh
#!/bin/bash
# додавання елементів масиву
function addarray {
  local sum=0
  local newarray
  newarray=(`echo "$@"`)
  for value in ${newarray[*]}
  do
    sum=$[ $sum + $value ]
  done
  echo $sum
}

myarray=(1 2 3 4 5)
echo "Оригінальний масив: ${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=`addarray $arg1`
echo "Сума елементів: $result"
```

```
$ ./test.sh
Оригінальний масив: 1 2 3 4 5
Сума елементів: 15
```

Передача елемента масиву із функції у сценарій виконується аналогічно. Функція використовує команду `echo` для виведення значень окремих елементів масиву, а в сценарії ці значення заносяться у нову змінну масив.

```
#!/bin/bash
# returning an array value
function arraydbl {
  local origarray
  local newarray
  local elements
  local i
  origarray=(`echo "$@"`) # локальний оригінальний масив у функції
  newarray=(`echo "$@"`)  # локальний новий масив у функції
  elements=$[ $# - 1 ]
  for (( i = 0; i <= $elements; i++ ))
  {
    newarray[$i]=$[ ${origarray[$i]} * 2 ]
  }
```

```
    echo ${newarray[*]}
  }

myarray=(1 2 3 4 5)
echo "Оригінальний масив: ${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=(`arraydbl $arg1`)  # створення нового массиву у сценарії
echo "Масив із функції: ${result[*]}"

$ ./test.sh
Оригінальний масив: 1 2 3 4 5
Масив із функції: 2 4 6 8 10
```

Сценарій передає значення масиву `myarray` використовуючи змінну `$arg1` у функцію `arraydblr`. Функція відновлює масив у нову змінну масив `origarray` і робить копію для вихідної змінної масиву `newarray` . Функція використовує команду `echo` для виведення значень змінної масиву `newarray`. Сценарій використовує вихід функції `arraydbl` для створення нового масиву `result`.

5 Рекурсія

Локальні змінні функцій мають властивість самовмістимості. Самовмістимі функції не використовують ніяких змінних поза функцією, крім тих що передаються як параметри з командного рядка. Ця властивість дозволяє функціям бути рекурсивними. *Рекурсивна функція* викликає сама себе для отримання результату. Звичайно рекурсивна функція має базове значення до якого вона ітеративно наближується. Так рекурсія `x! = x * (x-1)!` обмежується значенням `x==1`:

```
# test.sh
#!/bin/bash
# using recursion
function factorial {
  if [ $1 -eq 1 ]
  then
    echo 1
  else
    local temp=$[ $1 - 1 ]
    local result=`factorial $temp`
    echo $[ $result * $1 ]
  fi
}

read -p "Ведіть значення: " value
result=`factorial $value`
echo "Факторіал $value: $result"

$ ./test.sh
Введіть значення: 5
Факторіал 5: 120
```

6 Бібліотечні файли функцій

Для виклику функцій у різних сценаріях оболонка Bash підтримує бібліотечні файли. Для цього потрібно створити окремий файл, наприклад з іменем `myfuncs`, який містить функції для використання в інших сценаріях:

```
# myfuncs
function addem {
echo $[ $1 + $2 ]
}

function multem {
```

```
echo $[ $1 * $2 ]
}
function divem {
if [ $2 -ne 0 ]
then
echo $[ $1 / $2 ]
else
echo -1
fi
}
```

Для підключення бібліотечного файлу до сценарію призначена команда `source`. Команда `source` має також скорочене позначення «**.**» , як інструкція крапка.

```
#!/bin/bash
# використання функцій з бібліотеки myfuncs
. ./myfuncs
value1=10
value2=5
result1=`addem $value1 $value2`
result2=`multem $value1 $value2`
result3=`divem $value1 $value2`
echo "Результат додавання: $result1"
echo "Результат множення: $result2"
echo "Результат ділення: $result3"
```

```
$ ./test.sh
Результат додавання: 15
Результат множення: 50
Результат ділення: 2
```

У сценарії успішно використані функції бібліотечного файлу `myfuncs`.

Недолік такого підходу у втраті визначення функцій при перевантаженні Bash оболонки. Це може бути суттєвим недоліком для складних сценаріїв. У таких випадках бібліотечний файл можна підключити у файлі `.bashrc` домашнього каталогу:

```
# .bashrc
# Source global definitions
if [ -r /etc/bashrc ]; then
   . /etc/bashrc
fi
   . /home/libraries/myfuncs
```

Тепер функції можна викликати з командного рядка

7 Сценарій демон

Демони (більшість серверів є демонами) – це програми, які виконуються незалежно від bash сесії і постійно виконують деяку задачу. Команда Linux `nohup` виконує команду так, що вона не завершується після від'єднання від сесії bash. Команда `nohup` понижує пріоритет виконуваної команди і перенаправляє стандартний потік виведення у файл `nohup.out` (так як сесія bash завершиться). Щоб уникнути створення файлу `nohup.out` потрібно закрити стандартний потік виведення або перенаправити його із кінцевого сценарію у сценарій демон. Так як `nohup` не запускає команди у фоновому режимі, то це має роботи кінцевий сценарій командою `&`.

Сценарій демон має нескінчений цикл в якому він перевіряє виконувану роботу. Для перевірки роботи демона використовується два методи: опитування або блокування. Демон виконується аж поки його не зупинять командами `suspend` або `kill`.

Висновки.

• Для повторного використання блоків сценаріїв Bash підтримує створення і виклик функцій.

• Для повернення значень із функцій використовується команда `return`.

• У функцію можна передати параметри через командний рядок. Якщо потрібно у функції використати параметри, які передаються в основний сценарій, то необхідно в сценарії вручну і явно передати ці параметри у функцію.

• У сценаріях і функціях можуть використовуватися глобальні і локальні змінні. Глобальні змінні доступні у сценаріях і функціях. Локальні змінні оголошуються і доступні тільки у всередині функції.

• Змінні масиви можна передавати і отримувати з функції як окремі значення.

• Bash функції підтримують рекурсивність.

• Bash сценарії можуть використовувати бібліотечні файли функцій з використанням команди `source`.

Література.

[1] C. Newham, B. Rosenblat. Learning the bash shell. Third Edition. O'Reilly, 2005. – 333 p.

[2]. Richurd Blum. Command line and shell scripting bible. Indianapolis, Indiana: Wiley Publishing. – 2008. – 809 p.

Запитання.

1. Які є формати синтаксису створення функцій?
2. Як викликати функцію і отримати код завершення функції.
3. Як отримати значення функції, а не цілочисельний код завершення?
4. Як передати параметри командного рядка сценарію у функцію?
5. Яке призначення змінних середовища `$@, $*, $#, $?.`
6. Використання глобальних і локальних змінних у сценаріях і функціях. Команди `local, declare.`
7. Як передати масив у функцію і як отримати з функції масив?
8. Особливості сценарію з рекурсивними функціями.
9. Створення і використання бібліотечного файлу функцій у сценаріях.
10. Сценарій демон, як він запускається, функціонує і завершується?

10. ПЕРЕНАПРАВЛЕННЯ ВВЕДЕННЯ-ВИВЕДЕННЯ У BASH

Мета. Вивчення перенаправлення введення-виведення даних у мові сценаріїв Bash

Вступ. Для введення-виведення даних використовуються стандартні потоки і файлові дескриптори. Процес в системі Linux може використовувати до 9-ти файлових дескрипторів. Можна задавати постійні і тимчасові перенаправлення потоків у Bash сценаріях. Отримати всю інформацію про відкриті файлові дескриптори всіх процесів Linux можна командою `lsof`.

В Linux існує спеціальний каталог `/tmp` для зберігання тимчасових файлів і каталогів. Тимчасові файли і каталоги користувач може створити командою `mktemp`.

План.
1 Пристрої файли і стандартні потоки
2 Стандартні файлові дескриптори і їх перенаправлення
3 Тимчасове і постійне перенаправлення
4 Створення власних перенаправлень
5 Створення дескрипторів для читання-записування
6 Закриття файлових дескрипторів
7 Інформація про файлові дескриптори

1 Пристрої файли і стандартні потоки

В Linux всі пристрої є файлами і зберігаються в каталозі `/dev`.
`/dev/stdin` – пристрій стандартного введення (дескриптор STDIN, 0)
`/dev/stdout` – пристрій стандартного виведення (дескриптор STDOUT, 1)
`/dev/stderr` – пристрій стандартного виведення помилок (дескриптор STDERR, 2)
`/dev/null` – пристрій утилізації любих даних які направлені на його вхід
`/dev/zero` – пристрій для створення файлів, які повністю заповнені нулями
`/dev/tty` – термінал або консоль в якій виконується програма
`/dev/dsp` – інтерфейс до пристроїв які відтворюють звук або звукової карти
`/dev/fd0` – перший гнучкий диск
`/dev/hda1` – перший розділ IDE диска
`/dev/sda1` – перший розділ SCSI диска

2 Стандартні файлові дескриптори і їх перенаправлення

В Linux кожний файл вважається об’єктом і ідентифікується файловим дескриптором (унікальним цілим числом, яке ідентифікує відкритий файл у консольній сесії). Кожний процес може мати до 9-ти дескрипторів відкритих файлів. Оболонка Bash резервує перші три файлові дескриптори (0, 1, 2) для оброблення введення і виведення сценаріїв.

Файловий дескриптор	Ім’я	Описання
0	STDIN	Стандартний потік введення консолі
1	STDOUT	Стандартний потік виведення консолі
2	STDERR	Стандартний потік виведення помилок

Файловий дескриптор STDIN посилається на потік введення з консолі (клавіатура). Багато Bash команд сприймають введення з консолі.

```
>cat # приймає введення з консолі
111  # введено
```

```
111  # виведено
222  # введено
222  # виведено
<CTRL+D>  # завершення введення
```

Використовуючи символ ″<″ можна переназначити файловий дескриптор введення консолі на дескриптор іншого файлу.

```
> cat < file   # введення даних з файлу
111            # виведення даних на екран
222
```

Файловий дескриптор STDOUT посилається на потік виведення консолі (екран). Використовуючи символ “>” можна перенаправити потік виведення у файл.

```
$ls –l > test # записування у файл результату виконання команди
$printf ″Повідомлення 1″ # стандартне виведення на екран
$printf ″Повідомлення 2″ > /dev/stdout  # виведення через стандартний пристрій
$printf ″Повідомлення 3″ > /dev/tty     # виведення на екран (безпосередньо)
```

Для подавлення виведення команди використовується нульовий пристрій Linux:

```
ls –l > /dev/null
```

Для добавлення даних у файл використовується символ ″>>″:

```
$ls –l >> test  # добавлення даних у файл
```

Перенаправлення даних з файлу на вхід команди:

```
command < file
$ wc < test6
2 3 6  # число рядків число слів число байтів
```

Синтаксис перенаправлення даних з консолі на вхід команди

```
command << символи початку даних
>дані
…
>дані
>символи кінця даних (мають співпадати із символами початку)
```

Приклад:

```
>wc << abc
>1
>2
>3
>abc
```

Для передачі даних з виходу однієї команди на вхід іншої використовуються канали (pipes):

```
command1 | command2
```

Так, виконання двох команд можна замінити одною командою

```
>rpm –qa > rpm.list
>sort < rpm.list
>rpm –qa | sort > rpm.list
```

Перенаправлення стандартних файлових дескрипторів виведення:

`1>  file1` – перенаправлення стандартного вихідного потоку `stdout` у файл

`1>> file1` – добавлення стандартного вихідного потоку `stdout` у файл

`2>  file2` – перенаправлення потоку помилок `stderr` у файл

`2>> file2` – добавлення потоку помилок `stderr` у файл

`&>  file3` - перенаправлення потоків `stdout` і `stderr` у файл

`&>> file3` - добавлення потоків `stdout` і `stderr` у файл

У файловий дескриптор потоку помилок STDERR направляються всі помилки Bash оболонки, а також команд, програм і сценаріїв, які в ній виконуються. За замовчуванням STDOUT і STDERR направляються на консоль.

Перенаправлення STDOUT і STDERR у різні файли:

```
$ls –al file xxx 1> test 2> err
```

Перенаправлення STDOUT і STDERR в один файл:

```
$ls –al file xxx &> test
```

Для створення реєстраційних файлів (log files) використовується команда `tee` (трійник, розгалуження). Вона дозволяє направити дані із STDIN у STDOUT (за замовчуванням) і вказаний файл за один раз

```
tee filename
$echo ”Повідомлення” | tee file # виведення в STDOUT і файл file
```

Команда `tee` також використовується з командою створення каналу для перенаправлення виходу любої команди

```
who | tee testfile
```

3 Тимчасове і постійне перенаправлення

Файлові дескриптори можуть бути перенаправлені:

– тимчасово для одноразового виконання команди;

– постійно для всіх команд сценарію.

Для *тимчасового* перенаправлення файлового дескриптора перед ним ставиться символ `&`:

```
echo ”Привіт”  >&1
echo ”Помилка” >&2
printf ”Повідомлення” >&1    # тимчасове стандартне виведення
$ ls -l incoming/orders      # записано у listing.txt
$ ls -l incoming/orders 1>&1 # записано у listing.txt
```

Для постійного перенаправлення файлового дескриптора використовується команда `exec`, яка запускає нову оболонку `Bash` і перенаправляє файловий дескриптор на час дії сценарію командою `exec`.

```
exec 1> testout  # перенаправлення всього виведення у файл
exec 2> err      # перенаправлення помилок у файл
```

Можна перенаправити стандартний потік введення STDIN з клавіатури на введення з файлу

```
exec 0< file
while read line
do
   echo $line
done
```

Команду `exec` можна використовувати багаторазово і перенаправляти вихід для різних файлових дескрипторів. Після перенаправлення можна завжди повернутися до стандартних дескрипторів.

4 Створення власних перенаправлень

Оболонка Bash дозволяє відкриття 9-ти файлових дескрипторів. Дескриптори з третього по дев’ятий можуть використовуватися для перенаправлення як введення, так і виведення.

Альтернативні файлові дескриптори можна призначити стандартним, а потім перепризначити стандартні альтернативним.

```
exec 3>&1
echo "Перенаправлення дескриптора 3 для 1" >&3
…
exec 1>&3
echo "Перенаправлення дескриптора 1 з 3" >&1
```

Аналогічно можна перепризначити STDIN у інший файловий дескриптор, а потім відновити стандартне значення STDIN.

```
exec 6<&0
```

```
exec 0< testfile
read line
echo $line
exec 0<&6
```

5 Створення дескрипторів для читання-записування

При читанні-записуванні одного файлу Bash підтримує внутрішній вказівник, який вказує на своє розміщення у файлі. Файлові дескриптори з можливістю читання-записування створюються командою `exec` <> з відповідним номером файлового дескриптора:

```
exec 3<> file
read line <&3
echo ”Test” >&3
```

6 Закриття файлових дескрипторів

Усі створені файлові дескриптори введення-виведення автоматично закриваються при виході із сценарію.

Для закриття файлового сценарію в середині сценарію, без виходу із нього, використовується спеціальна стрічка `&-`:

```
exec 3>&-    # закриття файлового дескриптора 3
```

Приклад.

```
exec 3> test
echo ”Повідомлення 1” >&3   # перенаправлення у дескриптор 3
echo 3>&-
echo ”Повідомлення 2” >&3   # помилка, дескриптор 3 закритий
```

7 Інформація про файлові дескриптори

Отримати інформацію про відкриті файлові дескриптори всіх процесів Linux дозволяє команда `lsof`. Для зменшення обсягу інформації при виведенні використовуються ключі:

```
-d 0,1,2,3       # вказують номери потрібних файлових дескрипторів
-p PID1 -p PID2  # вказують PID потрібних процесів
-p $$            # PID поточного процесу
-a –p $$ -d 0,1,2, # дозволяє з використанням AND об’єднувати інші операції
```

8 Використання тимчасових файлів і каталогів

В Linux існує спеціальний каталог `/tmp` для розміщення тимчасових файлів. Любий користувач має право доступу для запису і читання в каталог `/tmp`. Систему можна налаштувати так, що тимчасові файли будуть знищуватися при завантаженні системи. Для створення тимчасових файлів і каталогів служить команда `mktemp` з відповідними ключами. Команда робить користувача власником файлу і надає тільки йому права на читання і записування (користувач `root` має права за замовчуванням).

Синтаксис команди `mktemp`

```
mktemp tmp.xx...x   #  10 символів після крапки
```

При запуску команди без параметрів розширення файлу `tmp` має унікальне випадкове значення.

Приклад створення тимчасового файлу і записування у нього даних:

```
tempfile =`mktemp tmp.a1b2c3d4e5’
exec 3>$tempfile
echo ”hello1” >&3
echo ”hello2’ >&3
exec 3>&-
```

```
cat $tempfile
rm -f $tempfile 2> /dev/null
```

Командою `mktemp` можна створювати як тимчасові файли так і тимчасові каталоги:

```
mktemp -t test.xxxxxxxxx
mktemp -d tmp.zzzzzzzzzz
```

Приклад створення тимчасового каталогу і файлу:

```
tempdir=`mktemp -d dir.xxxxxxxxxx`
cd $tempdir
f1=`mktemp -f tmp.xxxxxxxxx1`
f2=`mktemp -f tmp.xxxxxxxxx2`
exec 7>f1
exec 8>f2
echo ”Hello1” >&7
echo ”hello2” >&8
```

Висновки.

• В Linux всі файли є пристроями і зберігаються у каталозі `/dev`.

• Файли ідентифікуються файловими дескрипторами.

• Кожний процес може мати до 9-ти дескрипторів відкритих файлів.

• Файлові дескриптори 0, 1, 2 з іменами STDIN, STDOUT, STDERR зарезервовані для введення і виведення у сценаріях.

• Файлові дескриптори можуть бути перенаправлені тимчасово для одноразового виконання команди або постійно для всіх команд сценарію.

• Дескриптори файлів з 3-го по 9-й можуть використовуватися для перенаправлення як введення, так і виведення.

• Для створення дескрипторів з можливістю читання-записування використовується команда `exec`.

• Інформацію про відкриті файлові дескриптори всіх процесів Linux дає команда `lsof`.

• Тимчасові файли і каталоги створюються командою `mktemp` і розміщується у системному каталозі `/tmp`.

Література.

[1] C. Newham, B. Rosenblat. Learning the bash shell. Third Edition. O’Reilly, 2005. – 333 p.

Запитання.

1. Вказати пристрої файли і дати їх характеристику.
2. Які імена, номери та призначення стандартних файлових дескрипторів.
3. Перенаправлення потоків з використаееям символів >, >>, <, <<, |.
4. Створення реєстраційних файлів.
5. Тимчасове перенаправлення файлових дескрипторів.
6. Постійне перенаправлення файлових дескрипторів.
7. Створення власних перенаправлень для потоків введення і виведення.
8. Створення дескрипторів з можливістю читання-записування у файли.
9. Отримання інформації про відкриті файлові дескриптори процесів Linux.
10. Створення і використання тимчасових каталогів і файлів.

11. ПРОЦЕСИ І СИГНАЛИ

Мета. Вивчення засобів роботи з процесами і сигналами

Вступ. Кожне задача при виконанні підтримується ОС як процес. Linux має команди для запуску і моніторингу виконуваних процесів. Одним із способів взаємодії процесів є використання сигналів. Процеси можуть відправляти і захоплювати сигнали

При запуску команд у командному інтерпретаторі він розпізнає їх як завдання.

План.

1. Процеси в Linux
2. Запуск процесів
3. Моніторинг процесів
4. Сигнали у Linux
5. Відправлення, генерація і захоплення сигналів
6. Завдання
7. Календарне керування запуском сценаріїв
8. Запуск сценаріїв на виконання з сценаріїв оболонки

1. Процеси в Linux

Кожний процес ОС Linux характеризується набором атрибутів, який відрізняє даний процес від всіх інших процесів. До таких атрибутів відносяться:

• *Ідентифікатор процесу (PID)* . Кожний процес в системі має унікальний ідентифікатор. Кожний новий запущений процес отримує номер на одиницю більше попереднього.

• *Ідентифікатор батьківського процесу (PPID).* Даний атрибут процес отримує під час свого запуску і використовується для отримання статусу батьківського процесу.

• *Реальний і ефективний ідентифікатори користувача (UID,EUID) і групи (GID, EGID)*. Дані атрибути процесу вказують про його належність до конкретного користувача і групи. Реальні ідентифікатори збігаються з ідентифікаторами користувача, який запустив процес, і групи, до якої він належить. Ефективні ідентифікатори вказують від чийого імені був запущений процес. Права доступу процесу до ресурсів ОС Linux визначаються ефективними ідентифікаторами. Якщо на виконуваному файлі програми встановлено спеціальний біт SGID або SUID, то процес даної програми буде володіти правами доступу власника виконуваного файлу. Для управління процесом (наприклад, `kill`) використовуються реальні ідентифікатори. Всі ідентифікатори передаються від батьківського до дочірнього процесу. Для перегляду даних атрибутів можна скористатися командою `ps`.

• *Пріоритет або динамічний пріоритет (priority) і відносний або статичний (nice) пріоритет процесу.* Статичний пріоритет або *nice*-пріоритет лежить в діапазоні від -20 до 19, а типово використовується значення 0. Значення -20 відповідає найбільш високому пріоритету, *nice*-пріоритет не змінюється планувальником, він успадковується від батьків або його вказує користувач. Динамічний пріоритет використовується планувальником для планування виконання процесів. Цей пріоритет зберігається в полі `prio` структури `task_struct` процесу. Динамічний пріоритет обчислюється виходячи зі значення параметра *nice* для даної задачі шляхом обчислення надбавки або штрафу, залежно від інтерактивності завдання. Користувач має можливість змінювати тільки статичний пріоритет процесу. При цьому підвищувати пріоритет може тільки *root* користувач В ОС Linux існують дві команди управління пріоритетом процесів: *nice* і *renice*.

• *Стан процесу***.** В ОС Linux кожен процес обов'язково знаходиться в одному з перерахованих нижче станів і може бути переведений з одного стану в інший системою або командами користувача. Розрізняють наступні стани процесів:

TASK_RUNNING – процес готовий до виконання або виконується (runnable). Позначається символом R.

TASK_INTERRUPTIBLE – очікуючий процес (sleeping). Цей стан означає, що процес ініціалізував виконання якоїсь системної операції і чекає її завершення. До таких операцій відносяться введення/виведення, завершення дочірнього процесу пі т.д. Процеси з таким станом позначаються символом S.

TASK_STOPPED – виконання процесу зупинено (stopping). Будь-який процес можна зупинити. Це може робити як система, так і користувач. Стан такого процесу позначається символом T.

TASK_ZOMBIE – завершений процес (zombie). Процеси даного стану виникають у випадку, коли батьківський процес не чекаючи завершення дочірнього процесу, продовжує паралельно працювати. Процеси з таким станом позначаються символом Z. Такі завершені процеси більше не виконуються системою, але далі продовжують утримувати апаратні ресурси (крім обчислювальних).

TASK_UNINTERRUPTIBLE – процес, який не переривається (uninterruptible). Процеси в цьому стані очікують завершення операції введення/виведення з прямим доступом до пам’яті. Такий процес не можна завершити, поки не завершиться операція введення/виведення. Процеси з таким станом позначаються символом D. Стан аналогічний *TASK_INTERRUPTIBLE*, за винятком того, що процес не відновлює виконання при отриманні сигналу. Використовується у випадку, коли процес повинен чекати безперервно або коли очікується, що певна подія може виникати досить часто. Так як завдання в цьому стані не відповідає на сигнали, *TASK_UNINTERRUPTIBLE* використовується не так часто, як *TASK_INTERRUPTIBLE*.

Типи процесів.

В Linux процеси поділяються на три типи:

• *Системні процеси* – є частиною ядра і завжди розміщені в оперативній пам'яті. Системні процеси не мають відповідних їм програм у вигляді виконуваних файлів і запускаються при ініціалізації ядра системи. Виконувані інструкції і дані цих процесів знаходяться в ядрі системи, тому вони можуть викликати функції і звертатися до даних, які недоступні для інших процесів. Системними процесами, наприклад, є: `shed` (диспетчер підкачки), `vhand` (диспетчер сторінкового заміщення), `kmadaemon` (диспетчер пам'яті ядра).

•*Демони* – це не інтерактивні процеси, які запускаються звичайним способом – шляхом завантаження в пам'ять відповідних їм програм (виконуваних файлів), і виконуються у фоновому режимі. Звичайно демони запускаються при ініціалізації системи (але після ініціалізації ядра) та забезпечують роботу різних підсистем: системи термінального доступу, системи друку, з системи мережевого доступу і мережевих послуг, поштовий сервер `dhcp` т. п. Демони не пов'язані ні з одним сеансом роботи користувача і не можуть безпосередньо керуватися користувачем. Більшу частину часу демони чекають поки той або інший процес запросить певну послугу, наприклад, доступ до файлового архіву або друк документу.

• *Прикладні (користувача) процеси* – це всі інші процеси, запущені в системі. Як правило, це процеси, породжені в рамках сеансу роботи користувача. Наприклад, команда `ls` породить відповідний процес такого типу. Найважливішим прикладним процесом є командний інтерпретатор (shell), який забезпечує роботу користувача в Linux. Він запускається відразу ж після реєстрації в системі. Прикладні процеси Linux можуть виконуватися як в інтерактивному, так і у фоновому режимі, але в будь-якому випадку час їх життя (і виконання) обмежений сеансом роботи користувача. При виході з системи всі прикладні процеси знищуються.

Ієрархія процесів.

В Linux реалізована чітка ієрархія процесів в системі. Кожний процес в системі має тільки одного батька і може мати один або більше породжених процесів. Фрагмент ієрархії процесів Linux показаний на рис.1.

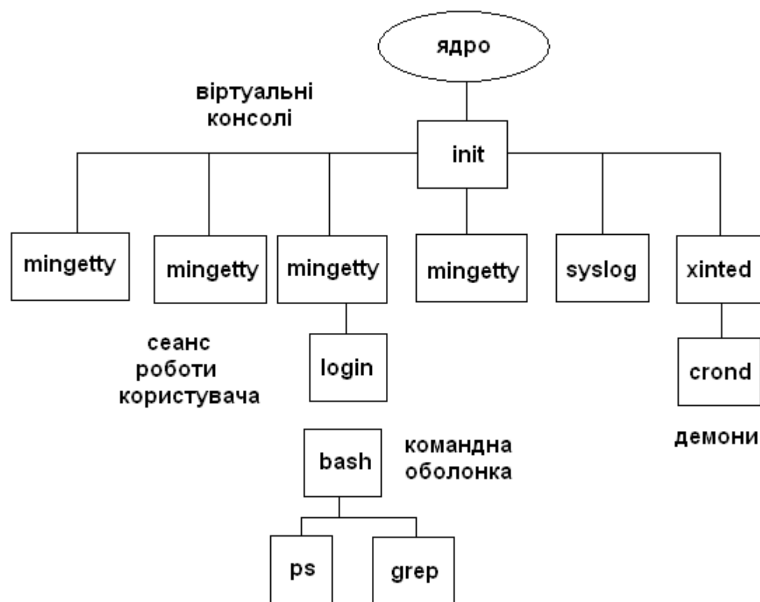

Рисунок1 – Фрагмент ієрархії процесів

На останній фазі завантаження ядро активує кореневу файлову систему і формує середовище виконання нульового процесу, створюючи простір процесу, ініціалізуючи нульову точку входу в таблиці процесу і роблячи кореневий каталог поточним для процесу. Коли формування середовища виконання процесу закінчується, система виконується вже як нульовий процес. Нульовий процес "галузиться", запускаючи програму `fork` прямо з ядра, оскільки сам процес виконується в режимі ядра. Код, який виконується породженим 1-м процесом, включає в себе виклик системної функції `exec`, запуск на виконання програми з файлу `/etc/init`. На відміну від 0-го процесу, який є процесом системного рівня, який виконуються в режимі ядра, 1-й процес відноситься до рівня користувача. Звичайно 1-й процес 1 іменується процесом `init`, оскільки він відповідає за ініціалізацію нових процесів. Можна помістити будь-яку програму в `/sbin/init` і ядро запустить її як тільки закінчить завантажуватися. Задачею `init` є запуск усіх інших потрібних програм.

Процес `init` читає файл `/etc/inittab`, в якому містяться інструкції для подальшої роботи. Першою інструкцією, звичайно, є запуск сценарію ініціалізації. У системах, заснованих на Debian, сценарієм ініціалізації буде `/etc/init.d/rcS`, у Red Hat – `/etc/rc.d/rc.sysinit`. Це те місце, де відбувається перевірка і монтування файлових систем (`/etc/fstab`), встановлення годин системного часу, включення розділу диску для підкачування, присвоєння імені хоста і т. п. Далі буде викликаний наступний сценарій, який переводить систему на "рівень запуску" за замовчуванням. Це означає деякий набір демонів, які повинні бути запущені.

`Syslogd` (`/etc/init.d/syslogd`) – сценарій, який відповідає за запуск і зупинку системи журнальної реєстрації подій SYSLOG, яка дозволяє записувати системні повідомлення у файли журналів /var/log.

`Xined` – демон Інтернет-служб, який керує сервісами для Інтернету. Демон прослуховує сокети і якщо в якомусь з них є повідомлення то визначає до якого сервісу належить даний сокет і викликає відповідну програму для оброблення запиту.

`crond` – демон `cron` відповідає за перегляд файлів `crontab` і виконання, внесених до нього команд у вказаний час для заданого користувача. Програма `crontab` взаємодіє з `crond` через файл `cron.update`, який повинен знаходитись разом з рештою файлів каталогу `crontab`, як правило у `/var/spool/cron/crontabs`.

Останньою важливою дією `init` є запуск деякої кількості `getty`. `Mingetty` – віртуальні термінали, які призначені для спостереження за консолями користувачів.

`getty` запускає програму `login` – початок сеансу роботи користувача в системі. Завданням `login` є реєстрація користувача в системі. Після успішної реєстрації найчастіше завантажується

командний інтерпретатор користувача (shell), який вказаний для даного користувача в файлі `/etc/passwd`.

2. Запуск процесів

Існує два способи запуску процесів в залежності від типу процесу. Процеси користувача запускаються в інтерактивному режимі шляхом введення команди або запуску сценарію. Для запуску системних процесів і демонів використовуються ініціалізаційні сценарії (`init`-сценарії). Такі сценарії використовуються процесом `init` для запусків інших процесів при завантаженні ОС. Ініціалізаційні сценарії зберігаються у каталозі `/etc`. У даному каталозі існують підкаталоги, іменовані rc0.d – rc6.d, кожний з яких асоційований з певним рівнем виконання (runlevel). У цих каталогів знаходяться символьні посилання на ініціалізаційні сценарії, які знаходяться в каталозі `/etc/rc.d/init.d`.

Слід зауважити, що в каталозі `/etc/init.d` присутні жорсткі посилання на сценарії каталогу `/etc/rc.d/init.d`, тому при зміні сценаріїв в цих каталогах змінені дані відображаються однаково незалежно від шляху до файлу сценарію.

Перегляд init-сценаріїв.

Всі init-сценарії можна повторно запускати або зупиняти, тим самим керуючи статусом сервісу, до якого вони належать. Сценарії запускаються з командного рядка за наступним синтаксисом:

```
/etc/init.d/script-name  start | stop | restart | condrestart | status | reload
```

Аргументи сценарію `script-name` можуть мати наступні значення:

- `start (запуск сервісу);`
- `stop (зупинка сервісу);`
- `restart (зупинка і подальший запуск сервісу);`
- `condrestart (умовна зупинка і наступний запуск сервісу);`
- `status (отримання статусу стану сервісу);`
- `reload (повторне зчитування конфігураційного файлу сервісу).`

Наприклад, для умовного перезапуску сервісу `sshd` використовується наступна команда:

```
[root@rhe!5 ~]# /etc/init.d/sshd condrestart
Stopping sshd: [ ok ]
Starting sshd: [ ok ]
```

При використанні аргументу `condrestart` перезапуск сервісу буде здійснено тільки у тому випадку, якщо сервіс вже працює у системі.

В ОС Linux для управління сервісами, крім безпосереднього звернення до файлу init-сценарію, існує спеціальна команда `service`, з аргументами аналогічними тим, що використовуються при безпосередньому запуску демонів через init-сценарії:

```
[root@rhel5 ~]# service sshd reload
```

Однак керувати демонами у більшості випадків може тільки користувач `root`.

Команди запуску процесів.

Команда `nice` запускає нові процеси із заданим пріоритетом:

```
nice -n 10 ./test4 > test4out &
```

Команда `renice` змінює пріоритети вже запущених процесів:

```
renice 10 -p 29504
```

3. Моніторинг процесів

Для перегляду запущених процесів в ОС Linux використовуються команди:
• `ps` – інтерактивно спостерігати за процесами (в реальному часі)
• `top` – вивести список процесів
• `uptime` – вивести завантаження системи
• `w` – вивести список активних процесів для всіх користувачів
• `free` – Вивести обсяг вільної пам'яті
• `pstree` – Відображає всі запущені процеси у вигляді ієрархії

Команда ps.

Якщо команду запустити без параметрів, то вона видає список процесів, запущених у поточному сеансі користувача. Приклади використання:

• `ps -A` – виводить список процесів з ідентифікаторами (PID) та їх іменами;
• `ps -ax` – виводить список процесів, з повним рядком запуску;
• `ps -U user` – список процесів породженим самим користувачем user;
• `ps T` – список задач, які пов'язані з поточним терміналом;
• `ps t ttyN` – список задач, пов'язаних з терміналом N;

Якщо список завдань великий, а нас цікавить стан однієї або кількох завдань, можна скористатися командою `grep`:

• `ps -U root | grep ppp` – виводить список завдань, які мають "ppp" в імені;

Команда top.

Для отримання відомостей про використання ресурсів комп'ютера можна скористатися командою `top`:

```
top - 17:17:13 up 28 min,  2 users,  load average: 0.19, 0.15, 0.18
Tasks: 178 total,   1 running, 177 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  0.2 sy,  0.0 ni, 99.3 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   4006392 total,  1696536 used,  2309856 free,     2096 buffers
KiB Swap:  2095100 total,        0 used,  2095100 free.   990828 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 2555 internet  20   0  637428  59260  48504 S 0.664 1.479   0:03.66 konsole
 1769 root      20   0  254768  68836  33232 S 0.332 1.718   0:19.37 X
 2357 internet  20   0 2224832 269008  98744 S 0.332 6.714   0:35.70 firefox
 2757 internet  20   0   15332   2724   2256 R 0.332 0.068   0:00.07 top
    1 root      20   0  119636   5972   4072 S 0.000 0.149   0:03.14 systemd
    2 root      20   0       0      0      0 S 0.000 0.000   0:00.00 kthreadd
    3 root      20   0       0      0      0 S 0.000 0.000   0:00.03 ksoftirqd/0
    5 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S 0.000 0.000   0:00.23 rcu_sched
    8 root      20   0       0      0      0 S 0.000 0.000   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S 0.000 0.000   0:00.00 migration/0
   10 root      rt   0       0      0      0 S 0.000 0.000   0:00.00 watchdog/0
   11 root      rt   0       0      0      0 S 0.000 0.000   0:00.00 watchdog/1
   12 root      rt   0       0      0      0 S 0.000 0.000   0:00.00 migration/1
   13 root      20   0       0      0      0 S 0.000 0.000   0:00.04 ksoftirqd/1
   15 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 kworker/1:0H
   17 root      20   0       0      0      0 S 0.000 0.000   0:00.00 kdevtmpfs
   18 root       0 -20       0      0      0 S 0.000 0.000   0:00.00 netns
```

У верхній частині вікна виводяться сумарні дані про стан системи – поточний час, час з моменту завантаження системи, число користувачів у системі, число процесів у різних станах, дані про використання процесора і пам'яті.

Нижче виводиться таблиця, що характеризує окремі процеси. Число рядків, що відображаються в цій таблиці, визначається розміром вікна. Вміст вікна оновлюється кожні 3

секунди. Натиснення клавіші `h` виводить довідку по командах, які дозволяють змінити формат даних що відображаються і управляти деякими параметрами процесів у системі:

- `s` або `d` - змінити інтервал оновлення вікна;
- `z` - кольорове/чорно-біле відображення;
- `n` або `#` - кількість процесів які відображаються;

Перемикати режими відображення можна за допомогою команд, які програма `top` сприймає. Це наступні команди):

- `Shift+N` - сортування по PID;
- `Shift+A` - сортувати процеси за віком;
- `Shift+P` - сортувати процеси щодо використання ЦП;
- `Shift+M` - сортувати процеси з використання пам'яті;
- `Shift+T` - сортування по часу виконання.

Однак, є й більш корисні команди, які дозволяють управляти процесами в інтерактивному режимі:

- `k` – зняти (kill) задачу. За цією командою буде відправлено запит на ідентифікатор процесу (PID), після введення якого, процес буде завершений.
- `r` – змінити поточний пріоритет завдання (renice). Як і при виконанні попередньої команди, буде запитаний PID, і потім, нове значення пріоритету (відображається в колонці NI).

Для виходу з команди `top` потрібно натиснути клавішу `q`.

Для більш наочного розуміння взаємозв'язку між процесами в ОС Linux існує команда `pstree`, яка відображає всі запущені процеси у вигляді ієрархії, за якою можна визначити взаємозв'язок між процесами.

4 Сигнали в Linux

Сигнали – це програмні переривання. Сигнали в ОС Linux використовуються як засоби синхронізації і взаємодії процесів. Сигнал є повідомленням, яке система посилає процесу або один процес посилає іншому. У Linux є команда `kill`, яка дозволяє послати заданому процесу будь-який сигнал. З точки зору користувача процесу отримання сигналу виглядає як виникнення переривання. Процес припиняє своє виконання, керування передається обробнику сигналу або ядру, якщо такого не має. Після закінчення оброблення сигналу процес може відновити своє виконання з тієї точки, де він був перерваний.

Кожний сигнал має ім’я та номер. Імена всіх сигналів починаються з послідовності SIG. Список сигналів можна отримати командою `kill -l`.

Таблиця 1 – Список сигналів

Сигнал	Мнемоніка	Описання
1	SIGHUP	Завершити процес (при завершенні поточної операції)
2	SIGINT	Перервати процес не виділяючи процесорний час (сигнал блокується і перехоплюється), `<CTRL+C>`
3	SIGQUIT	Завершити процес (як SIGTERM) і вивести дамп пам’яті
4	SIGILL	Недійсна інструкція (not reset)
5	SIGTRAP	Трасування пастки (trace trap)
6	SIGABRT	Вихід (abort)
7	SIGBUS	Помилка шини (Bus error)
8	SIGFPE	Виняткова ситуація з плаваючою крапкою
9	SIGKILL	Безумовно завершити процес (не блокується і не перехоплюється)
10	SIGUSR1	Визначено користувачем

11	SIGSEGV	Порушення сегментації
12	SIGUSR2	Визначено користувачем
13	SIGPIPE	Запис у канал з неможливістю зчитування
14	SIGALRM	Генерується таймером, встановленим функцією `alarm()`
15	SIGTERM	При можливості коректно завершити процес (блокується і перехоплюється)
16	SIGSTKFLT	
17	SIGCHLD	Зміна статусу нащадка
18	SIGCONT	Продовжити зупинений процес
19	SIGSTOP	Безумовно зупинити, але не завершувати процес
20	SIGTSTP	Зробити паузу або зупинити процес (без завершення), `<CTRL+Z>`
21	SIGTTIN	Спроба фонового tty на читання
22	SIGTTOU	Спроба фонового tty на запис
23	SIGURG	Термінова умова на каналі вводу/виводу
24	SIGXCPU	Вичерпано ліміт часу ЦП
25	SIGXFSZ	Перевищено ліміт розміру файлу
26	SIGVTALRM	Вичерпаний час віртуального таймера
27	SIGPROF	Вичерпаний час таймера профілю
28	SIGWINCH	Змінено розмір вікна
29	SIGIO	Можливий ввід/вивід
30	SIGPWR	Помилка живлення
31	SIGSYS	Помилковий системний виклик
34	SIGRTMIN	
35	SIGRTMIN+1	
	…	
49	SIGRTMIN+15	
50	SIGRTMAX-14	
	…	
63	SIGRTMAX-1	
64	SIGRTMAX	

Linux підтримує 31 сигнал (номери від 1 до 31). Мнемонічні імена, які починаються з приставки SIG (SIGTERM, SIGINT, SIGKILL) використовуються у мовах програмування таких як Сі. В інтерпретаторі bash використовуються або числа або мнемонічні імена, але без приставки SIG – TERM, INT, KILL.

Сигнали можуть породжуватися різними умовами:

1. Генеруватися терміналом, при натисканні певної комбінації клавіш, наприклад, натискання Ctrl+C генерує сигнал SIGINT, таким чином можна перервати виконання програми, яка вийшла з під контролю;
2. Апаратні помилки (ділення на 0, помилка доступу до пам'яті та інші) також генерують сигнали. Ці помилки звичайно виявляються апаратним забезпеченням, яка повідомляє про них ядру. Після цього ядро генерує відповідний сигнал і передає його процесу, який виконувався в момент появи помилки. Наприклад, сигнал SIGSEGV надсилається процесу у разі спроби звернення за неправильною адресою в пам'яті.
3. Іншим процесом (у тому числі і ядром і системним процесом), який виконав системний виклик `kill()`;
4. При виконанні команди `kill`.

При отриманні сигналу процес може запросити ядро виконати одну з трьох реакцій на сигнал:

1. Примусово проігнорувати сигнал (практично будь-який сигнал може бути проігноровано, крім SIGKILL і SIGSTOP).

2. Обробити сигнал за замовчуванням: проігнорувати, зупинити процес, перевести в стан очікування до отримання другого спеціального сигналу або завершити роботу.

3. Перехопити сигнал (виконати оброблення сигналу, задане користувачем).

5. Відправлення, генерація і захоплення сигналів

Відправлення сигналів.

Процеси можуть взаємодіяти між собою посилаючи сигнали. *Сигнал процесу* – це повідомлення, яке сигнал розпізнає і може ігнорувати або реагувати на нього.

В Linux є дві команди, які дозволяють послати сигнал запущеному на виконання процесу. За замовчуванням команда `kill PID` посилає сигнал `TERM` процесу із заданим ідентифікаторами `PID`. Сигнал `TERM` "дружньо" повідомляє процес про необхідність завершення. Але якщо процес в даний момент виконується, він найбільш ймовірно проігнорує повідомлення. У цьому випадку йому можна відправити більш "строгі" повідомлення INT або HUP. Якщо процес розпізнає ці сигнали, він завершить виконання поточної операції, а потім завершиться сам, наприклад:

```
kill –s HUP 3940
```

Найбільш "строгим" є сигнал KILL. При його отриманні процес негайно завершує роботу

```
kill –s KILL 3940
```

Команда `killall` дозволяє зупиняти процеси за їхніми іменами, а не ідентифікаторами.

За замовчуванням оболонка Bash ігнорує сигнали `SIGQUIT (3)` і `SIGTERM (15)` для того, щоб під час інтерактивної взаємодії її не можна було випадково завершити. Але вона обробляє сигнали `SIGHUP(1)` і `SIGINT (2)`. При отриманні сигналу `SIGHUP` оболонка передає його усім процесам, які в ній виконуються і потім завершує роботу. При отриманні сигналу `SIGINT` оболонка тільки перериває своє виконання. Ядро Linux перестає надавати оболонці процесорний час. Коли це трапляється оболонка в свою чергу направляє цей сигнал усім своїм процесам.

Оболонка також надсилає ці сигнали і сценаріям, які в ній виконуються. За замовчування сценарії ігнорують ці сигнали, але в них при потребі можна обробити надіслані сигнали.

Генерація сигналів.

Оболонка Bash дозволяє генерувати два базових Linux сигнали використовуючи комбінацію клавіш клавіатури. `Ctrl-C` комбінація генерує сигнал SIGINT і посилає його всім поточним процесам оболонки і перериває їх виконання.

`Ctrl-Z` комбінація генерує сигнал SIGTSTP, який зупиняє всі процеси в оболонці. Перелік зупинених процесів можна продивитися командою `ps au`. Завершити роботу оболонки із зупиненими процесами дозволяє команда `exit`. При завершенні роботи оболонки завершується робота і всіх зупинених процесів.

Захоплення сигналів.

Сценарії можуть захоплювати сигнали командою `trap`. Формат команди

```
trap команда сигнал
```

Приклад перехоплення сигналів:

```
trap echo "Перехоплено сигнали" SIGINT SIGTERM
count=1
while [ $count –le 10 ]
do
  echo "Цикл: $count"
  sleep 5
  count = $[ $count + 1 ]
done
```

При виконанні такий сценарій нечутливий до натискання комбінації клавіш `Ctrl-C`.

Приклад перехоплення завершення роботи сценарію:

```
trap echo ”Вихід” EXIT
count=1
while [ $count –le 5 ]
do
  echo ”Цикл: $count”
  sleep 3
  count = $[ $count + 1 ]
done
```

Відміна дії команди `trap`:

```
trap – EXIT
```

Приклад:

```
trap echo ”Вихід” EXIT
count=1
while [ $count –le 5 ]
do
  echo ”Цикл: $count”
  sleep 3
  count = $[ $count + 1 ]
done
trap - Exit
```

6. Завдання

Завдання (job) – це просто робоча одиниця командного процесора. При запуску команди, поточний командний інтерпретатор визначає її як завдання і слідкує за нею. Коли команда виконана, відповідне завдання зникає. Завдання знаходяться на більш високому рівні, ніж процеси. Операційна система Linux нічого про них не знає. Вони є лише елементами командного процесора.

До команд управління завданнями відносяться:

- `jobs` – вивести список завдань;
- `&` – виконати завдання у фоновому режимі;
- `Ctrl+Z` – призупинити виконання поточного (інтерактивного) завдання;
- `suspend` – призупинити командний процесор;
- `fg` – перевести завдання в інтерактивний режим виконання;
- `bg` – перевести призупинене завдання у фоновий режим виконання.

У фоновому режимі роботи сценарій не асоціюється із стандартними потоками STDIN, STDOUT, STDERR термінальної сесії. Для запуску сценарію у фоновому режимі потрібно розмістити символ `&` після команди:

```
./test &
```

Коли сценарій виконується у фоновому режимі, він не займає консоль і в ній можна запускати інші команди або сценарії. Але при завершенні роботи консолі всі процеси, в тому числі і фонові завершуються. Для продовження роботи фонових процесів, після завершення роботи консолі, використовується команда `nohup`, яка блокує любі `SIGHUP` сигнали, послані процесу.

```
$nohup ./test &
```

Команда оболонка назначає звичайним і фоновим процесам номери завдань (job number), а Linux назначає ідентифікатори процесів PID. При використанні команди `nohub` сценарій ігнорує сигнал `SIGHUB`, який надсилається оболонкою при її закритті.

Так як команда `nohub` деасоціює процес від терміналу, то процес втрачає зв'язки з STDOUT і STDERR. Тому команда перенаправляє усі повідомлення для STDOUT і STDERR у файл `nohup.out`.

Для отримання переліку завдань, які виконуються у поточній сесії оболонки, використовується команда `jobs`.

Параметри команди `jobs`:

`-l` – список завдань із PID та `job` номерами;

`-n` – список завдань, які змінили свій статус;

`-p` – списик завдань із їх PID;

`-r` – список завдань, які виконуються;

`-s` – список зупинених завдань.

Любе зупинене завдання в основному чи фоновому режимах можна запустити на продовження виконання. У фоновому режимі командою `bg номер_завдання`, а в основному режимі – командою `fg номер_завдання`.

7 Календарне керування запуском сценаріїв

Для керування запуском сценаріїв в часі призначені команди `at`, `batch`, `cron` програма і таблиця, `anacron` програма.

Команда `at` дозволяє задати час коли Linux запустить сценарій на виконання. Для цього вона направляє завдання з часовою позначкою у чергу. Інша команда `atd (at demon)`, працюючи у фоновому режимі, перевіряє кожні 60 сек часові позначки завдань у черзі (каталог `/var/spool/at`) і при співпадінні з поточним часом запускає їх на виконання. Команда `atd` запускається при завантаженні Linux.

Формат команди `at`:

```
at [-f filename] time
```

`-f filename` – розміщення файлу із сценарієм;

`time` – час запуску завдання на виконання.

Черга складається з 26 підчерг з різними пріоритетами, які мають позначки від `a` до `z`. За замовчування всі завдання направляються у чергу з найвищим пріоритетом `a`. Чергу з нижчим пріоритетом можна задати ключем `-q`.

Коли завдання виконується у Linux, воно не має асоційованої з не командної консолі. Тому Linux направляє повідомлення про виконання завдання на `e-mail`, а не STDOUT і STDERR.

Для роботи з чергує є ряд команд:

`atq` – список завдань в черзі на виконання

`atrm номер_завдання` – вилучення завдання із черги

Команда `batch` дозволяє запускати завдання на виконання при низькому завантаженні системи. Команда перевіряє завантаженість Linux системи і якщо вона менша 0.8, то вона запускає на виконання завдання із черги.

Формат команди `batch`:

```
batch [-f filename] time
```

`-f filename` – розміщення файлу із сценарієм;

`time` – час, після якого завдання можна пробувати виконати.

Програма `cron` дозволяє запускати завдання на виконання регулярно за календарем. Програма виконується у фоновому режимі і перевіряє `cron` таблицю. `Cron` таблиця має спеціальний формат, який дозволяє задати календарну регулярність виконання завдання. Формат таблиці `cron`:

```
хвилини години день_місяця  місяць день_тижня команда
```

Наприклад, виконання сценарію `test` кожного дня в 10 год 15 хв:

```
15 10 * * * /home/user/test > test.out
```

Виконати сценарій в 12 год дня останнього дня місяця:

```
00 12 * * * if [ `date +%d -d tomorrow` = 01 ] ; then ; command
```

Кожний користувач може створити свою `cron` таблицю для керування виконанням сценаріїв. Для створення і роботи з `cron` таблицею призначена команда `crontab`. При роботі з `cron` таблицею передбачається, що Linux буде безперервно працювати увесь календарний період. Якщо Linux деякий час не буде працювати, то і завдання в цей час не виконаються. Для розв'язання цієї проблеми призначена програма `anacron`.

Програма anacron використовує часові позначки для визначення того, чи завдання було виконано. Якщо деяке завдання позначене як не виконане, то воно буде при можливості якнайшвидше запущене на виконання. Програма *anacron* використовує свою таблицю, розміщену в `/etc/anacrontab`, для описання завдань. Формат таблиці:

```
період затримка ідентифікатор команда
```

`період` – задає як часто завдання запускається на виконання, в днях

`затримка` – затримка на виконання сценарію, після старту програми `anacron`

`ідентифікатор` – стрічка, яка унікально ідентифікує завдання у повідомленнях

8 Запуск сценаріїв на виконання з сценаріїв оболонки

Сценарії можна автоматично запускати на виконання *при завантаженні системи Linux*. При цьому у пам'ять завантажується ядро Linux і запускається перший процес `/sbin/init`, `PID` якого дорівнює 1. `Init` процес запускає усі інші процеси в системі.

`Init` процес читає файл `/etc/inittab`, який містить сценарії, які запускає `init` програма для різних рівнів виконання `Linux`, які описані в таблиці 1.

Таблиця 2 – Рівні виконання Linux

Рівень виконання	Описання
0	Вимкнення
1	Режим одного користувача
2	Режим багатьох користувачів без мережі
3	Режим багатьох користувачів з мережею
4	Не використовується
5	Режим багатьох користувачів з мережею і графічною сесією X Window.
6	Перевантаження

Сценарій `rc` визначає рівень виконання Linux і запускає на виконання відповідні сценарії. Застосування запускаються на виконання сценарієм `startup scripts`. Цей сценарій, в залежності від дистрибутива Linux, може розміщуватися в `/etc/rc.d`, `/etc/init.d` або в `/etc/init.d/rc.d`.

Розміщення локальних сценаріїв, які запускаються при завантаженні системи, також залежить від дистрибутива Linux:

```
Debian    /etc/init.d/rc.local
Fedora    /etc/rc.d/rc.local
OpenSuse  /etc/init.d/boot.local
```

Сценарії можна запускати на виконання *при завантаженні нової командної оболонки.* Кожний `home` каталог користувача містить два файли `.bash_profile` і `.bashrc`, які використовує Bash оболонка для автоматичного запуску сценаріїв і встановлення змінних середовища. Сценарій `.bash_profile` запускається на виконання при реєстрації нового користувача і відповідно запуску нової командної оболонки. Тому тут можна розміщувати сценарії користувача, які виконуються при його реєстрації в системі.

Оболонка Bash виконує сценарій `bash.rc` при запуску кожної нової командної оболонки. Тому в цьому файлі можна розмістити локальний сценарій, який буде виконуватися при запуску нової командної оболонки поточним користувачем системи.

Якщо розмістити сценарій у файлі `/etc/bashrc`, то він буде виконуватися при запуску командної оболонки любим користувачем системи.

Висновки.

• Linux використовує понад 30 сигналів для взаємодії з процесами.

• Найбільше використовуються сигнали для зупинки, переривання, продовження і завершення процесів.

• Процеси також можуть взаємодіяти між собою посилаючи сигнали.

• Оболонка Bash дозволяє генерувати два базових сигнали використовуючи комбінацію клавіш `Ctrl-C` (сигнал SIGINT) і `Ctrl-Z` (сигнал SIGTSTP).

• Сценарії Bash можуть перехоплювати сигнали командою `trap`.

• Сценарії Bash можуть виконуватися в основному або фоновому режимах.

• Змінити пріоритети виконуваних команд і сценаріїв може команда `nice`.

• Для часового керування сценаріями призначені команди `at`, `batch`, `cron` програма і таблиця, `anacron` програма.

• Сценарії на виконання можна запускати при завантаженні системи Linux або нової командної оболонки.

Література.

[1] C. Newham, B. Rosenblat. Learning the bash shell. Third Edition. O'Reilly, 2005. – 333 p.

[2]. Richurd Blum. Command line and shell scripting bible. Indianapolis, Indiana: Wiley Publishing. – 2008. – 809 p.

Запитання.

1. Стани і типи процесів.
2. Ієрархія процесів у Linux.
3. Запуск і моніторинг процесів у Linux.
4. Сигнали у Linux.
5. Яка різниця між сигналами завершення процесів SIGHUP, SIGKILL, SIGTERM.
6. За допомогою якої команди можна послати сигнали виконуваному процесу.
7. Які сигнали може генерувати оболонка Bash.
8. Як запустити і зупинити завдання в основному і фоновому режимі.
9. Як змінити пріоритет процесу при запуску і виконанні.
10. Призначення і формат команди `at`.
11. Призначення і формат команди `batch`.
12. Призначення і формат команд `cron` і `anacron`.
13. Як забезпечити запуск сценаріїв при завантаженні системи Linux і нової командної оболонки.

12. ТЕКСТОВІ СПИСКИ ВИБОРУ У BASH. КОМАНДА DIALOG

Мета. Вивчення засобів створення текстових списків вибору, керування кольорами дисплею, застосування команди `dialog` для створення діалогових вікон.

Вступ. Взаємодія із сценаріями можлива не тільки за допомогою команд `echo` і `read`. У Bash сценаріях можна використовувати текстові списки вибору, керувати кольорами дисплею та вбудовувати діалогові вікна.

План.
1 Створення текстових списків вибору (меню)
2 Використання команди `select`
3 Керування кольорами із сценаріїв
4 Використання кольорів у сценаріях
5 Використання віконних віджетів у сценаріях
6 Використання команди `dialog` у сценаріях

1 Створення текстових списків вибору (меню)

Звичайно для створення інтерактивних сценаріїв використовують списки вибору (меню). Меню містить список доступних опцій, перенумерованих буквами або цифрами, з яких користувач може зробити вибір. Текстові списки вибору можна створити з використанням команди оболонки `case`.

Перед створенням текстового списку вибору (меню) очищається екран командою `clear`. Потім можна використати декілька команд `echo` з опцією `–e`, яка дозволяє виводити на екран не тільки текст, але і керуючі послідовності символів `\n`, `\t` та інші.

```
clear
echo
echo -e "\t\t\tМеню СисАдміна\n"
echo -e "\t1. Вивести дисковий простір"
echo -e "\t2. Вивести зареєстрованих користувачів"
echo -e "\t3. Вивести інформацію про використання пам’яті"
echo -e "\t0. Вийти з меню\n\n"
echo -en "\t\tЗробіть вибір: "
```

Остання команда `echo` має опцією `–en`, яка подавляє виведення з нового рядка. Це дозволяє наступній команді зчитати дані з поточного рядка. Для забезпечення затримки і зчитування даних з однієї позиції без натискання клавіші `Enter` використовується команда `read –n 1`. Послідовність списку для вибору можна оформити як одну функцію `function menu`.

```
function menu {
  clear
  echo
  echo -e "\t\t\tМеню СисАдміна\n"
  echo -e "\t1. Вивести дисковий простір"
  echo -e "\t2. Вивести зареєстрованих користувачів"
  echo -e "\t3. Вивести інформацію про використання пам’яті"
  echo -e "\t0. Вийти з меню\n\n"
  echo -en "\t\tЗробити вибір: "
  read –n 1 option
}
```

При виборі кожного пункту із списку має викликатися відповідна функція, наприклад `diskspace`, `whoseon`, `memusage`. Логіку такого вибору можна організувати на основі команди `case`.

```
menu
case $option in
```

```
0)
   break ;;
1)
   diskspace ;;
2)
   whoseon ;;
3)
   memusage ;;
*)
   clear
   echo "Вибачте, невірний вибір";;
esac
```

Для забезпечення повторюваності виборів команда `case` поміщається у нескінчений цикл `while [ 1 ] do … done`. Тоді повністю зібраний сценарій списку вибору матиме наступний вигляд:

```
#!/bin/bash
function diskspace {
  clear
  df -k
}
function whoseon {
  clear
  who
}
function memusage {
  clear
  cat /proc/meminfo
}
function menu {
  clear
  echo
  echo -e "\t\t\tМеню СисАдміна\n"
  echo -e "\t1. Вивести дисковий простір"
  echo -e "\t2. Вивести зареєстрованих користувачів"
  echo -e "\t3. Вивести інформацію про використання пам’яті"
  echo -e "\t0. Вийти з меню\n\n"
  echo -en "\t\tЗробити вибір: "
  read –n 1 option
}

while [ 1 ]
do
  menu
  case $option in
  0)
     break ;;
  1)
     diskspace ;;
  2)
     whoseon ;;
  3)
     memusage ;;
  *)
     clear
     echo "Вибачте, невірний вибір";;
  esac
done
clear
```

Список має три вибори в яких викликаються відповідні функції, а вибір 0 забезпечує вихід із списку. Використовуючи такий шаблон можна створити любі сценарії із списком вибору.

2 Використання команди select

Команда `select` дозволяє створювати меню в одному рядку і автоматично вводити і обробляти відповіді. Формат команди `select`:

```
select variable in list
do
  commands
done
```

де `list` список елементів меню відокремлених пропусками, `PS3` – змінна середовища, значення якої виводиться у кінці перенумерованого списку елементів меню.

Приклад команди `select`:

```
#!/bin/bash
# using select in the menu
function diskspace {
  clear
  df -k
}

function whoseon {
  clear
  who
}

function memusage {
  clear
  cat /proc/meminfo
}
PS3=”Enter option: ”
select option in ”Display disk space” ”Display logged on users”
”Display memory usage” ”Exit program”
do
   case $option in
   ”Exit program”)
       break ;;
   ”Display disk space”)
       diskspace ;;
   ”Display logged on users”)
       whoseon ;;
   ”Display memory usage”)
       memusage ;;
   *)
       clear
       echo ”Sorry, wrong selection”;;
  esac
done
clear
```

При виконанні сценарію появиться наступне меню:

```
$ ./smenu1
1) Display disk space 3) Display memory usage
2) Display logged on users 4) Exit program
Enter option:
```

3 Керування кольорами із сценаріїв

Більшість програм емуляторів терміналів розпізнають керуючі ANSI `Esc`-символи. ANSI `Esc`-символи починаються з керуючої послідовності CSI (control sequence indicator), яка вказує,

що за нею розміщуються параметри, які встановлюють режим відображення дисплею та задають колір тексту і фону.

Режимом відображення дисплею задають параметри SGR (Select Graphic Rendition). Синтаксис параметрів SGR:

`CSIn[;k]m`

`CSI` – керуючі `Esc`-символи;
`n`, `k` параметри, які визначають режим відображення дисплею;
`m` – признак SGR параметрів.
Можна задати один або два параметри одночасно, розділивши їх символом `';'`
Значення параметрів для керування режимом відображення дисплею показані в табл. 1.

Таблиця 1

Параметри SGR для керування режимом відображення

Парам.	Описання
0	Скинути у нормальний режим
1	Встановити інтенсивність `bold`
2	Встановити інтенсивність `faint`
3	Використати шрифт `italic`
4	Використати одиночне підкреслення
5	Встановити повільне блимання
6	Встановити швидке блимання
7	Інвертувати кольори тексту/фону
8	Встановити колір основного тексту у колір фону

Приклади керування режимами відображення дисплею:
`CSI3m` – відображення шрифтів `italic`;
`CSI3;5m` – відображення шрифтів `italic` з повільним блиманням.

Для керування кольором тексту і фону використовуються коди ANSI кольорів показані у табл. 2.

Таблиця 2

Коди ANSI кольорів

Код	Описання
0	Чорний
1	Червоний
2	Зелений
3	Жовтий
4	Голубий
5	Бордовий
6	Бірюзовий
7	Білий

При заданні кольору тексту і фону використовують подвійні цифри. Для тексту вказується перша цифра 3, а для фону – 4. Друга цифра задає код кольору.

Приклади:

`CSI37m` – білий текст
`CSI47m` – білий фон

Можна об’єднати колір тексту і фону, задавши їх через символ `";"`:

CSI31;40m – текст червоний, фон – чорний.

ANSI Esc-символи можна послати у сесії терміналу командою echo. Код CSI складається з послідовності двох символів: Esc-символу (^[ ) і символу [. Esc-символ у більшості редакторів використовується для інших потреб. Тому для генерування Esc-символу в редакторах використовується послідовність натискання клавіш Ctrl-v, а потім Esc. В результаті появиться символ **^[**. Таким чином отриманий код CSI можна перевірити у командному рядку:

```
>echo ^[[41mThis is a test^[[40m
```

У команді printf escape символ (Esc - **^[**) можна задати послідовністю символів '\e', наприклад:

```
printf "RGB кольори \e[31m R \e[0m \e[32m R \e[0m \e[34m R \e[0m \n"
printf "\e[1m Жирний \e[0m \e[3m Похилий \e[0m \e[4m Підкресл \e[0m \n"
printf "\e[31;1m Червоний жирний \e[0m \n"
printf "\e[37;4;2m Білий, підкреслений, слабо інтенсивний \e[0m \n"
```

Зміна кольору стрічки повідомлення командного рядка:

```
$ PS1='\e[1;34m\u@h:\w\e[0m\$'
```

де \e[1;34m - встановлення синього кольору тексту;
\u – user;
\h – host;
\w – повний шлях (\W – остання частина повного шляху).

Для зміни кольорів тексту і фону можна використати команду tput:
tput setaf color – задати колір тексту;
tput setaf color – задати колір фону;
tput smul – режим підкреслення;
tput usmul – відмінити режим підкреслення;
tput bold – режим жирного тексту;
tput dim – режим зменшеної яскравості;
tput sgr0 – скинути всі атрибути і режими;

4 Використання кольорів у сценаріях

Керуючі Esc-символи можна використовувати у сценаріях. Але необхідно мати на увазі, що коли емулятор терміналу бачить ці коди, то він їх і обробляє. Так cat команди виводить символи на дисплей, які потім інтерпретуються емулятором терміналу, змінюючи режими відображення дисплею.

Приклад сценарію меню, в якому використовуються Esc-коди для керування кольором:

```
#!/bin/bash
# menu using colors
function diskspace {
  clear
  df -k
}

function whoseon {
  clear
  who
}

function memusage {
  clear
```

```
  cat /proc/meminfo
}

function menu {
  clear
  echo
  echo -e "\t\t\tSys Admin Menu\n"
  echo -e "\t1. Display disk space"
  echo -e "\t2. Display logged on users"
  echo -e "\t3. Display memory usage"
  echo -e "^[[1m\t0. Exit program\n\n^[[0m^[[44;33m"
  echo -en "\t\tEnter option: "
  read -n 1 option
}

echo "^[[44;33m"
while [ 1 ]
  do
    menu
    case $option in
    0)
       break ;;
    1)
       diskspace ;;
    2)
       whoseon ;;
    3)
       memusage ;;
    *)
    clear
    echo -e "^[[5m\t\t\tSorry, wrong selection^[[0m^[[44;33m";;
    esac
    echo -en "\n\n\t\t\tHit any key to continue"
    read -n 1 line
done
echo "^[[0m"
clear
```

Меню появляється у жовтому тексті на голубому фоні.

5 Використання віконних віджетів у сценаріях

Пакет `dialog` створює стандартні діалогові вікна у текстовому режимі з використанням ANSI escape керуючих кодів. Ці діалогові вікна можна вбудовувати у сценарії для взаємодії з користувачем. Команда `dialog` використовує параметри командного рядка для задання необхідного типу вікна діалогу. Формат команди:

```
dialog --widget parameters
```

де `widget` – ім’я віджету, табл. 3;
`parameters` – розміри віджету та любий текст необхідний для віджету.

Таблиця 3

Діалогові віджети

calendar	Календар з вибором дати
checklist	Багатократний вибір (позначками)
form	Форма з текстовими полями і надписами
fselect	Вікно вибору файлу
gauge	Індикатор з процентом виконання
infobox	Вікно повідомлення без очікування відповіді

inputbox	Форма з текстовим полем для введення
inputmenu	Меню з можливістю редагування
menu	Список вибору
msgbox	Видача повідомлення з кнопкою Ok
pause	Індикатор паузи
passwordbox	Поле паролю
passwordform	Форма з полем паролю
radiolist	Радіокнопки
tailbox	Висвітлення текстового файлу у вікно командою tail
tailboxbg	Висвітлення текстового файлу у вікно командою tail у фоновому режимі
textbox	Висвітлення тестового файлу у вікно з прокруткою
timebox	Вікно з вибором годин, хвилин, секунд
yesno	Повідомлення з кнопками Yes і No

Кожний діалоговий віджет підтримує виведення у двох формах:
- з використанням `STDERR`;
- з використанням `exit` коду.

`Exit` код команди `dialog` визначається кнопкою, яку натиснув користувач. Для кнопок `OK`, `Yes` `exit` код дорівнює 0, а для кнопок `Cancel`, `No` – дорівнює 1. Можна використати стандартну змінну `$?` для визначення натиснутої кнопки у діалоговому віджеті.

Якщо віджет повертає любі дані, наприклад, вибір меню, то команда `dialog` надсилає їх у потік STDERR. Можна перенаправити потік STDERR у інший файл, наприклад:

```
dialog --inputbox "Enter your age:" 10 20 2>age.txt
```

Приклади використання команд dialog:

```
dialog --msgbox text height width
$ dialog --title Testing --msgbox "This is a test" 10 20

$ dialog --title "Please answer" --yesno "Is this thing on?" 10 20
$ echo $?
1

$ dialog --inputbox "Enter your age:" 10 20 2>age.txt
$ echo $?
0
$ cat age.txt
12$

$ dialog --textbox /etc/passwd 15 45

$ dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space"
2 "Display users" 3 "Display memory usage" 4 "Exit" 2> text.txt

$ dialog --title "Select a file" --fselect $HOME/ 10 50 2>file.txt
```

Команда `dialog` має великий набір опцій, які дозволяють налаштувати вигляд і властивості діалогових вікон, табл. 4.

Таблиця 4

Оції команди dialog

Опції	Описання
--add-widget	Продовжити наступний діалог, незважаючи на натиснення клавіші Esc або кнопки Cancel
--aspect *ratio*	Задати відношення ширина/висота вікна
--backtitle *title*	Задати надпис зверху вікна на задньому фоні
--begin *x y*	Задати координати верхнього-лівого кута вікна
--cancel-label *label*	Задати альтернативний надпис у кнопці Cancel
--clear	Очистити дисплей використовуючи колір

	заднього фону вікна діалогу
--colors	Дозволити вбудовувати ANSI коди кольорів і діалог тексту
--cr-wrap	Дозволити символ ”\n” у діалозі тексту і здійснювати перенесення рядка
--create-rc *file*	Записати взірець конфігураційного файлу у заданий файл
--defaultno	Зробити за замовчуванням No у діалозі Yes/No
--default-item *string*	Встановити значення за замовчуванням у checklist, form, menu діалогах
--exit-label *label*	Задати альтернативну позначку в кнопці Exit
--extra-button	Висвіти додаткову кнопку між кнопками Ok, Cancel
--extra-label *label*	Задати альтернативний надпис у кнопці Extra
--help	Висвітити help повідомлення
--help-button	Висвітити кнопку Help після кнопок Ok і Cancel
--help-label *label*	Задати альтернативний надпис у кнопці Help
--help-status	Записати інформацію з checklist, radiolist або form після help інформації при натисканні кнопки Help
--ignore	Ігнорувати нерозпізнані опції команди dialog
--input-fd *fd*	Задати альтернативний файловий дескриптор, інший ніж STDIN
--insecure	Змінити віджет password для висвічування зірочок
--item-help	Добавити стовпчики внизу дисплея для кожного тега в checklist, radiolist або menu
--keep-window	Не чистити старі віджети з екрану
--max-input *size*	Задати максимальний розмір стрічки при введенні (за замовчуванням 2048)
--nocancel	Подавити кнопку Cancel
--no-collapse	Не конвертувати табуляцію у пропуски і діалозі тексту
--no-kill	Розмістити діалог tailbox у задньому фоні і заборонити SIGHUP для процесів
--no-label *label*	Задати альтернативний надпис у кнопці No
--no-shadow	Не висвічувати тіні для вікон діалогу
--ok-label *label*	Задати альтернативний надпис у кнопці Ok
--output-fd *fd*	Задати альтернативний дескриптор вихідного файлу, інший ніж STDERR
--print-maxsize	Надрукувати максимальний розмір діалогового вікна дозволений у вивід
--print-size	Надрукувати розмір кожного діалогового вікна дозволений у вивід
--print-version	Надрукувати версію dialog у вивід
--separate-output	Вивести результат віджету checklist в один рядок без лапок
--separator *string*	Задати стрічку, яка розділятиме виведення кожного віджету
--separate-widget*string*	Задати стрічку, яка розділятиме виведення кожного віджету
--shadow	Малювати тіні знизу і справа кожного вікна
--single-quoted	Використовувати одинарні лапки для виведення cheklist
--sleep *sec*	Затримка на задане число секунд після обробки діалового вікна
--stderr	Направити виведення на STDERR
--stdout	Направити виведення на STDOUT
--tab-correct	Конвертувати табуляції у пропуски
--tab-len *n*	Задати число пропусків у табуляції (за замовчування 8)

--timeout *sec*	Задати число секунд перед виходом з кодом помилки при відсутності введення користувача
--title *title*	Задати заголовок вікна діалогу
--trim	Вилучити пропуски і "\n" з тексту діалогу
--visit-items	Модифікувати зупинку табуляції у вікні діалога для включення списку елементів
--yes-label *label*	Задати альтернативний надпис у кнопці Yes

6 Використання команди `dialog` у сценаріях

При використанні команди `dialog` потрібно виконати наступне:
- перевірити `exit` код команди `dialog`, якщо є кнопки `Cancel` або `No`;
- перенаправити STDERR для отримання вихідних значень.

Приклад реалізацію меню з використання команди `dialog`:

```
#!/bin/bash
# using dialog to create a menu
temp=`mktemp -t test.XXXXXX`
temp2=`mktemp -t test2.XXXXXX`
function diskspace {
  df -k > $temp
  dialog --textbox $temp 20 60
}
function whoseon {
  who > $temp
  dialog --textbox $temp 20 50
}

function memusage {
  cat /proc/meminfo > $temp
  dialog --textbox $temp 20 50
}

while [ 1 ]
do
dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space" 2
"Display users" 3 "Display memory usage" 0 "Exit" 2> $temp2
if [ $? -eq 1 ]
then
  break
fi
selection=`cat $temp2`
case $selection in
1)
  diskspace ;;
2)
  whoseon ;;
3)
  memusage ;;
0)
  break ;;
  *)
   dialog --msgbox "Sorry, invalid selection" 10 30
  esac
  done
    rm -f $temp  2> /dev/null
    rm -f $temp2 2> /dev/null
```

Сценарій використовує нескінчений цикл `while` для висвітлення меню діалогу. Вікно команди `dialog` має кнопку `Cancel`. Тому в циклі перевіряється `exit` код команди `dialog`. При

натисканні на кнопку `Cancel` повертається код 1, за яким по команді `break` здійснюється вихід із циклу.

Сценарій використовує команду `mktemp` для створення двох тимчасових файлів, в яких містяться дані команди `dialog`. Файл `$temp` містить вихід команди `df`, а файл `$temp2` – вибране значення меню.

Висновки.

• Сценарії Bash дозволяють створювати текстові меню. Команда `select` дозволяє створювати меню в одному рядку і автоматично вводити і обробляти відповіді.

• Для керування кольорами дисплею використовуються керуючі ANSI escape коди.

• У сценаріях можна вбудовувати діалогові вікна з використанням команди `dialog`.

Література.

[1] C. Newham, B. Rosenblat. Learning the bash shell. Third Edition. O'Reilly, 2005. – 333 p.

[2]. Richurd Blum. Command line and shell scripting bible. Indianapolis, Indiana: Wiley Publishing. – 2008. – 809 p.

Запитання.

1. За допомогою яких команд Bash можна створювати текстові меню.
2. Які переваги команди `select` при створенні текстових меню.
3. Як міняти кольори і режими роботи дисплею з використанням керуючих ANSI `Esc`-символів за допомогою команди `echo`.
4. Як міняти кольори і режими роботи дисплею з використанням керуючих ANSI `Esc`-символів за допомогою команди `printf`.
5. Як міняти кольори і режими роботи дисплею з використанням керуючих ANSI `Esc`-символів за допомогою команди `tput`.
6. Як створюються віконні діалоги за допомогою команди `dialog`.

13. ПОТОКОВІ РЕДАКТОРИ SED I GAWK

Мета. Вивчення можливостей і застосування у сценаріях Bash потокових редакторів `sed` і `gawk`.

Вступ. Використання потокових редакторів `sed` і `gawk` значно доповнює можливості bash сценаріїв у перевірці даних, створенні звітів, індексуванні текстів, видобуванні бітів і байтів з текстових файлів для подальшого їх оброблення, сортування даних.

План.
1 Короткі теоретичні відомості
2 Потоковий редактор `sed`
3 Розширений потоковий редактор `awk/gawk`
3.1 Пошук за шаблоном
3.2 Використання регулярних виразів
3.3 Введення і виведення даних
3.4 Вирази, змінні і операції
3.5 Масиви
3.6 Функції

1. Короткі теоретичні відомості

Крім звичайний інтерактивних редакторів тексту існують і потокові редактори. Потокові редактори редагують текст на основі попередньо записаних правил. В Linux набули поширення два потокових редактори: `sed` і `gawk`.

2. Потоковий редактор sed

Потоковий редактор `sed` маніпулює даними у потоці даних на основі команд введених у командному рядку або у командному текстовому файлі. Він читає один рядок даних із стандартного входу `STDIN`, порівнює дані із заданими командами редактора, змінює дані згідно заданих команд і виводить новий рядок у `STDOUT`. Таким чином обробляються усі рядки даних і редактор завершує роботу. Формат виклику потокового редактора `sed`:

```
sed [options] {command1 command2…} file
sed [-n][-e] 'command' file(s)
sed [-n] –f gawk_script file(s)
```

`options` – опції команди `sed`:

`-e 'command'` – додає команди задані в командному рядку (якщо більше одної), до команд які обробляють вхід;

`-f gawk_script` – додає команди задані в файлі, до команд які обробляють вхід;

`-n` – не виводить вихід кожної команди, але чекає на команду `print`.

`{command1 command2 …}` – задаються окремі команди, які застосовуються до одної адреси і вводяться з нового рядка у командному рядку.

`file` – файл, до якого застосовуються команди.

`Sed` не модифікує дані у вхідному файлі, а застосовує команди до даних вхідного потоку `STDIN`, який направляється на вивід. Це дозволяє направляти дані через канал на вхід редактору `sed`

```
$ echo "This is a test" | sed 's/test/big test/'
```

У прикладі, дані направляються через канал на вхід потокового редактора де до них застосовується команда s, яка замінює перший рядок другим test->big.

Для запуску sed з читанням команд із файлу потрібно команди записати у файл script1, а дані – у файл data1

```
$ cat script1
s/test/big test/
```

```
$ cat data1
This is a test
```

```
$ sed -f script1 data1
$ cat data1
This is a big test
```

Потоковий редактор sed має велику кількість команд і форматів ($ sed --help) для редагування даних.

Команда підставлення даних **s** (substitude):

```
[address]s/pattern/replacement/flags
```

підставляє замість шаблону *pattern* заміну *replacement*,

прапор *flags:*

- N – число, яке вказує на кількість підставлень;
- g – виконати всі підставлення;
- p – друк вмісту файлу шаблона;
- w file – вивести у файл результат підставлень.

Замість розділювача стрічок ”/” можна використовувати “!” (у випадку наявності в тексті символів /):

```
$ sed ’s!/bin/bash!/bin/csh!’ /etc/passwd
```

Підставлення, в 2-му рядку, в 2- і 3-му рядку, з 2-го рядка і до кінця тексту:

```
$ sed ’2s/dog/cat/’ data1
$ sed ’2,3s/dog/cat/’ data1
$ sed ’2,$s/dog/cat/’ data1
```

Два варіанти запису декількох команд у командному рядку:

```
$ sed -e ’s/brown/green/; s/dog/cat/’ data1
$ sed –e ’
  >s/brown/green/
  >s/dog/cat/’ data1
```

Використання шаблону *pattern* для фільтрування тексту. Команда підставлення буде застосована тільки до тих рядків, які містять текст шаблону /*pattern*/command.

Команда буде застосована до тих рядків файлу, які містять слово rich:

```
$ sed ’/rich/s/bash/csh/’ /etc/passwd
```

Команди підставлення можна згрупувати, використовуючи фігурні дужки n,m{} і застосувати до рядків від n до m:

```
$ sed ’2,7{
> s/fox/elephant/
> s/dog/cat/
> }’ data1
```

Рядки із заданими номерами n,m вилучаються командою n,m**d** (delete). Вилучення 2-го рядку, 2- і 3-го рядка, 2-го рядка і до кінця тексту:

```
sed ’3d’   data2
```

```
sed '2,3d' data2
sed '3,$d' data2
```

Рядки із заданим шаблоном `/pattern/` вилучаються командою `/pattern/d`. Вилучення порожніх рядків:

```
sed /^$/d file > new_file
```

Вилучення рядків, які починаються або закінчуються на xxx:

```
sed '/^xxx/d; /xxx$/d' file > new_file
```

Команда **i** (insert) вставляє новий рядок перед заданим рядком, а команда **a** (append) – після заданого рядка:

```
sed '[address]command\new line'  file

$ sed '3i\Цей рядок вставляється перед 3-м рядком' data2
$ sed '3a\Цей рядок всталяється після 3-го рядка' data2
```

Команда заміни всього рядка за номером або за шаблоном **c** (change):

```
sed '[address]c\new line'  file
$ sed '3c\Цей рядок замінить третій рядок' data2
$ sed '/abcd/c\Цей рядок замінить рядок за шаблоном' data2
```

Команда **y** (translate) для заданих рядків послідовно замінює символи з набору `inchars` у набір `outchars`

```
[address]y/inchars/outchars/
$ sed 'y/123/789/' data7
$ echo "ABCDEFGH" | sed 'y/ABCDEFGH/12345678/'
```

Для друкування інформації з потоку даних використовуються наступні команди:
p – друк рядків тексту;
= – друк номерів рядків;
l – друк тексту і недрукованих (невидимих) символів ASCII як двоцифрових кодів.

Команда **p** друкує задані рядки (-n подавити вивід решти рядків):

```
$ echo "Тестовий рядок" | sed 'p'
$ sed -n '/number 3/p' data2
$ sed -n '2,3p' data2
$ sed -n '/3/{p s/line/test/p }' data2
```

Друк усіх рядків з номерами і друк рядків за шаблоном

```
$ sed '=' data1
$ sed -n '/number 4/{ = p }' data2
```

Друк рядків з недрукованими ASCII символами

```
$ sed -n 'l' data2
```

Команди запису **w** (write) і читання **r** (read) файлів

```
[address]w filename
[address]r filename
```

Записати у файл 1-й і 2-й рядок із вхідного потоку:

```
$ sed '1,2w test' data6
```

Прочитати дані з одного файлу і вставити їх в інший файл:

```
$ sed '3r data1' data2   # після 3-го рядка файлу data2 вставити файл data1
sed '/number 2/r data1' data2 # після рядків з шаблоном вставити файл data1
sed '$r data1' data2    # вставити файл data1 в кінці файла data2

$ cat letter
```

```
Наступним працівникам:
LIST
надіслати повідомлення

$ cat data3
Іваненко І.І.
Петренко П.П.
Степаненко С.С

$ sed '/LIST/{
> r data3
> d                # вилучення шаблону LIST
> }' letter
```

Або в один рядок

```
$ sed /LIST/'r data' letter | sed /LIST/d
```

```
Наступним працівникам:
Іваненко І.І.
Петренко П.П.
Степаненко С.С
надіслати повідомлення
```

3. Розширений потоковий редактор awk/gawk

Розширений потоковий редактор gawk призначений для пошуку рядків (або стрічок) у файлах, які співпадають із заданим шаблоном, і виконання заданих дій із такими рядками. Редактор gawk, використовує для роботи з текстом не команди редактора, а мову програмування. Ця мова є орієнтована на дані, тобто спочатку необхідно вказати, які дані потрібно вибрати з файлу, а потім над знайденими даними виконати необхідні дії. Тому програма gawk складається з правил, які звичайно записуються в окремих рядках:

```
шаблон  { дія }
шаблон  { дія }
…
```

Програму gawk можна запускати різними способами. Якщо програма коротка – її можна записати в командному рядку:

```
awk 'program' input-file1 input-file2 ...
```

Якщо програма велика – її записують в окремий файл, з якого вона зчитується на виконання:

```
awk -f program-file input-file1 input-file2 ...
```

Програму gawk можна оформити як автономний сценарій:

```
# test1
#!/bin/gawk -f

BEGIN { print "Привіт від gawk" }
```

і запустити на виконання

```
$ chmod +x test1
$ ./test1
Привіт від gawk
```

В рамках цієї мови програмування можна:
- визначити змінні для зберігання тексту ($0, $1,…);
- використати арифметичні і стрічкові операції для маніпулювання даними;
- писати структуровані програми;
- генерувати форматовані звіти з вхідного файлу у вихідний в іншому порядку і форматі.

Програма `gawk` використовує внутрішні змінні для роботи з полями даних одного рядка:

`$0` – містить увесь текст одного рядка;
`$1` – містить перше поле рядка тексту;
`$2` – містить друге поле рядка тексту;
…
`$n` – містить `n`-е поле рядка тексту;

Вивід першого поля з кожного рядка тексту

```
$ cat data3 або gawk '{print}' data3
One line of test text.
Two lines of test text.
Three lines of test text.
$ gawk '{print $1}' data3
One
Two
Three
```

При читанні файлів можна використовувати різні розділювачі полів:

```
$ gawk -F: '{print $1}' /etc/passwd
at
avahi
bin
daemon
dnsmasq
ftp
ftpsecure
games
lp
```

В програмі можна використовувати декілька команд:

```
$ echo "My name is Petro" | gawk '{$4="Ivan"; print $0}'
My name is Ivan
```

Читання програми з файлу:

```
$ cat script2
{ print $5 "'s userid is " $1 }
$ gawk -F: -f script2 /etc/passwd
Batch jobs daemon user id is at
User for Avavhi user id is avahi
bin user id is bin
Daemon user id is daemon
Dns dnsmasq user id is dnsmasq
FTP account user id is ftp
```

Якщо є декілька команд у сценарії, то команди починаються з нового рядка (без ;)

```
$ cat script3
{
text="'s userid is "
print $5 text $1
}
$ gawk -F: -f script3 /etc/passwd | more
root's userid is root
bin's userid is bin
```

Виконання команд програми перед і після оброблення даних

```
$ echo "1111" | gawk 'BEGIN { print "Hello word!"} { print $0 }'
```

```
Hello word!
1111

$ echo "1111" | gawk 'BEGIN {print "Hello World!"} {print $0} END {print
"byebye"}'
Hello word!
1111
byebye
```

Читання команд програми gawk з файлу:

```
$ cat script4
BEGIN {
print "The latest list of users and shells"
print " Userid Shell"
print "-------- -------"
FS=":"
}
{
print $1 " " $7
}
END {
print "This concludes the listing"
}

$ gawk -f script4 /etc/passwd
Userid Shell
------ -----
at /bin/bash
avahi /bin/false
daemon /bin/bash
...
This concludes the listing
```

3.1 Пошук за шаблоном

Програму qawk можна записати у файл, наприклад у test1:

```
# програма test1, яка знаходить лексеми integer, letter, blank line
/[0-9]+/    { print "That is an integer" }
/[A-Za-z]+/ { print "This is a string" }
/^$/        { print "This is a blank line." }
```

Приклад використання програми, яка обробляє вхід з консолі:

```
$ gawk -f test1
4
That is an integer
t
This is a string
4T
That is an integer
This is a string
RETURN
This is a blank line.
44
That is an integer
CTRL-D
$
```

3.2 Використання регулярних виразів

Регулярні вирази можна використовувати як шаблон між двома похилими. Наступний приклад друкує з файлу `file` 2-ге поле кожного запису, який містить слово `'Hello'`:

```
$ gawk '/Hello/ { print $2 }' file
```

Регулярні вирази можуть використовуватися у порівняннях. Для порівняння з регулярним виразом використовуються оператори `'~'`, `'!~'`. Наступні приклади вибирають всі записи, у яких перше поле починається з букви *J*, не починається із букви *J*, містить цифри

```
$ gawk '$1 ~ /J/' file
$ gawk '$1 !~ /J/' file
$ gawk 'BEGIN {digit = "[0-9]+" } $1 ~ digit { print }' file
```

3.3 Введення і виведення даних

`Gawk` при введенні даних розбиває їх на записи і поля. Кількість прочитаних записів із вхідного файлу зберігається у вбудованій змінній FNR. Записи розділюються символом розділювачем, який задається у вбудованій змінній RS. Приклад розділення на записи з використанням символу розділювача '/'.

```
gawk 'BEGIN { RS = '/' } { print }' file
```

Записи розділюються на поля з використання символу пропуску. Символ розділювач полів задається у вбудованій змінній `FS`. Для посилання на поля використовуються змінні `$1`, `$2`, … . Змінна `$0` посилається на весь запис.

Значення полів можна корегувати

```
gawk '{ S1 = $1 - 10; $2 = $3 + $4 } { FS = ':'; print }' file
```

Для виведення даних використовується команда `print`:

```
gawk 'BEGIN { print "A      B"
              print "-------"}
            { print  $1, $2 } file
```

Кожна команда `print` створює вихідний запис. Розділювачем вихідних записів є вбудована змінна `ORS` (за замовчуванням `ORS='\n'`). Розділювачем вихідних полів є вбудована змінна `OFS`.

Для форматованого виведення значень полів використовується команда printf:

```
printf "format" var1, var2, …
```

3.4 Вирази, змінні і операції

Вирази є базовими будівельними блоками шаблонів і дій. Вирази будуються із констант, змінних і операцій над ними. Найпростішим типом виразів є константи: числа (всі числа в `gawk` є числами з *плаваючою крапкою*), стрічки, регулярні вирази. Для зберігання значень їх присвоюють змінним *variable = text*.

Перетворення типів стрічка-число виконується в контексті `gawk` програми.

```
two =2; three =3
print (two three) + 4
```

Результатом виконання програми буде 27.

Логічним типом є `true` і `false`. У `gawk` любі ненульові числові значення і непорожні стрічки мають значення `true`. Всі інші значення є `false`.

У виразах використовуються арифметичні і стрічкові операції. Арифметичні операції:

```
x ^ y, x ** y x в ступені y
-x – унарний мінус
+x – унарний плюс
x*y - множення
x/y – ділення без округлення ( '3/4' має значення 0.75 )
```

```
x%y – залишок від ділення.
x+y - додавання.
++x - інкремент
x-y - віднімання.
--x - декремент
```

Є тільки одна стрічкова операція – зчеплення. Для цього один вираз записується безпосередньо біля другого:

```
$ awk '{ print "Field number one: " $1 }' file
```

Операція порівняння:

```
x < y True якщо x менше y.
x <= y True якщо x менше або дорівнює y.
x > y True якщо x більше y.
x >= y True якщо x більше або дорівнює y.
x == y True якщо x рівне y.
x != y True якщо x не рівне y.
x ~ y True якщо x спіпадає з regexp позначеним як y.
x !~ y True якщо x не співпадає з regexp позначеним як y.
subscript in array True якщо array має елемент subscript.
```

Конструкції галуження:

```
if (x % 2 == 0)
  print "x парне"
else
  print "x непарне"


$ echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
false


switch (NR * 2 + 1) {
case 3:
case "11":
    print NR - 1
    break

case /2[[:digit:]]+/:
    print NR

default:
    print NR + 1

case -1:
    print NR * -1
}
```

Конструкції циклів:

```
gawk '{
  i = 1
  while (i <= 3) {
    print $i
    i++
  }
}' file
```

```
gawk '{
  i = 1
  do {
    print $0
    i++
  } while (i <= 10)
}' file
```

```
awk '{
for (i = 1; i <= 3; i++)
    print $i
}' file
```

Всередині конструкцій циклів можуть використовуватися інструкції `break`, `continue`, `next`, `nextfile`, `exit`.

3.5 Масиви

Масиви `gawk` є асоціативними, тобто елемент масиву складається з індекса і значення. Індексом може бути як число, так і стрічка.

```
Index 3 Value 30
Index 1 Value "foo"
Index 0 Value 8
Index 2 Value ""
```

```
Index "dog" Value "chien"
Index "cat" Value "chat"
Index "one" Value "un"
Index 1 Value "un"
```

Доступ до елементів масиву:

```
{
  if ($1 > max)
    max = $1
    arr[$1] = $0
}

END {
for (x = 1; x <= max; x++)
  if (x in arr)
    print arr[x]
}
```

```
{
  for (i = 1; i <= NF; i++)
  used[$i] = 1
}

END {
  for (x in used) {
     if (length(x) > 10) {
        ++num_long_words
        print x
     }
  }
  print num_long_words, " > 10 chars"
}
```

3.6 Функції

В `gawk` використовуються вбудовані і визначені користувачем функції.

Вбудовані числові функції: `atan2(y, x)`, `cos(x)`, `exp(x)`, `int(x)`, `log(x)`, `rand()`,`sin(x)`, `sqrt(x)`, `srand([x])`.

Функції для маніпуляції стрічками: `asort(source [, dest [, how ] ])`, `asorti(source [, dest [, how ] ])`, `gensub(regexp, replacement, how [, target])`, `gsub(regexp, replacement [, target])`, `index(in, find)`, `length([string])`, `match(string, regexp [, array])`, `patsplit(string, array [, fieldpat [, seps ] ])` , `split(string, array [, fieldsep [, seps ] ])`, `sprintf(format, expression1, ...)`, `strtonum(str)`, `sub(regexp, replacement [, target])`, `substr(string, start [, length])`, `tolower(string)`, `toupper(string)`.

Функції введення/виведення: `close(filename [, how])`, `fflush([filename])`, `system(command)`.

Функції роботи з часом: `mktime(datespec)`, `strftime([format [, timestamp [, utc-flag]]])`, `systime()`.

Функції маніпуляції з бітами: `and(v1, v2 [, . . . ])`, `compl(val)`, `lshift(val, count)`, `or(v1, v2 [, . . . ])`, `rshift(val, count)`, `xor(v1, v2 [, . . . ])`.

Функції визначені користувачем:

```
function name([parameter-list])
{
   body-of-function
}
```

```
function myprint(num)
{
   printf "%6.3g\n", num
}
if ($3 > 0) { myprint($3) }
```

Висновки.

• У сценарії Bash можна використовувати потокові редактори `sed` і `gawk`.

• Потоковий редактор `sed` маніпулює даними у потоці даних на основі команд введених у командному рядку або у командному текстовому файлі.

• Розширений потоковий редактор `gawk` призначений для пошуку рядків (або стрічок) у файлах, які співпадають із заданим шаблоном, і виконання заданих дій із такими рядками. Редактор `gawk`, використовує для роботи з текстом не команди редактора, а мову програмування.

Література.

[1] Arnold D. Robbins. GAWK: Effective gawk programming. A User's Guide for GNU Awk, Edition 4.1. Free Software Faundation, Inc. 2013. – 490 p.

[2] Dale Dougherty, Arnold Robbins. Sed & awk. O'Really, Second Edition, 1997. – 432 p.

Запитання.

1. Яка відмінність потокових редакторів від звичайних і чому їх використовують у сценаріях Bash.
2. Команди потокового редактора `sed` підставлення `s`, вилучення `d`, вставлення `i`, заміни символів з набору `y`, друкування `p`, запису `w`.
3. Особливості введення і виведення даних в потоковому редакторі `gawk`.
4. Вирази, змінні і операції в потоковому редакторі `gawk`.

СПИСОК ЛІТЕРАТУРИ

1. Гордеев В.В., Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2001. – 736.

2. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 576 с.

3. Блум Ричард, Бреснахэн Кристина. Командная строчка Linux и сценарии оболочки. Библия пользователя, 2-е узд.: Пер. с англ. – М.: ООО “И.Д. Вильямс”, 2012. – 784 с.

4. Джонсон, Майкл К., Троан, Єрик В. Разработка приложений в среде Linux, 2-е изд.: Пер. с англ. –М.: “ООО И.Д. Вильямс”, 2007. – 544 с.

5. С.Л. Скловская. Команды Linux. – СПб.: Питер, 2004. – 848 с.

6. Митчелл М., Оулдем Д., Самьюєл А. Программирование для Linux. Профессиональный поход. – М.: Вильямс, 2002. – 288 с.

7. C. Newham, B. Rosenblat. Learning the bash shell. Third Edition. O’Reilly, 2005. – 333 p.

8. Richurd Blum. Command line and shell scripting bible. Indianapolis, Indiana: Wiley Publishing. – 2008. – 809 p.

9. Arnold D. Robbins. GAWK: Effective gawk programming. A User’s Guide for GNU Awk, Edition 4.1. Free Software Faundation, Inc. 2013. – 490 p.

10. Dale Dougherty, Arnold Robbins. Sed & awk. O’Really, Second Edition, 1997. – 432 p.