МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
«Прикарпатський національний університет
імені Василя Стефаника»

Фізико-технічний факультет

Голота В. І.

Курс лекцій з дисципліни

“СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ. МОВА СЦЕНАРІЇВ BASH"

Електронне мережеве навчальне видання

Івано-Франківськ, 2024

УДК 004.432.2
Г 61

Рекомендовано до друку Вченою радою фізико-технічного факультету Прикарпатського національного університету імені Василя Стефаника (протокол № 2 від 24 жовтня 2024 року)

Рецензенти:

Когут Ігор Тимофійович, завідувач кафедри комп'ютерної інженерії та електроніки Прикарпатського національного університету імені Василя Стефаника, професор, доктор технічних наук

Никируй Любомир Іванович, завідувач кафедри фізики і хімії твердого тіла Прикарпатського національного університету імені Василя Стефаника, професор, кандидат фізико-математичних наук

Курс лекцій з дисципліни "Системне програмне забезпечення. Мова сценаріїв Bash". [Електронний ресурс] / Прикарпатський національний університет імені Василя Стефаника; уклад. Голота В. І. – Електронні текстові дані (1 файл: 10 Мбайт). – Івано-Франківськ: Прикарпатський національний університет імені Василя Стефаника, 2024. – 245 с. – Назва з екрану.

У курсі лекцій розглянуто команди ОС Linux, консольні і потокові редактори, оболонку Bash і її систему розширення, оператори, функції, перенаправлення стандартних потоків введення-виведення, процеси, сигнали, інтерактивні і мережеві команди.

Курс лекцій призначений для здобувачів ступеня бакалавра галузі знань 12 "Інформаційні технології" спеціальності 123 "Комп'ютерна інженерія". Може бути також корисним студентам, які вивчають системне програмне забезпечення у курсах споріднених дисциплін відповідних спеціальностей.

УДК 004.42(076.6)

ЗМІСТ

1. ОСНОВИ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Мета. Ознайомити студентів із основами побудови 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. Моделі розроблення програмного забезпечення
7. Повторне використання коду
8. 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 часто використовуваних операцій 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 при відключеній системі переривань. Модуль виконується до свого кінця, після чого він може бути знову викликаний на виконання з іншого завдання (процесу). Привілейований модуль функціонує як почергово або 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.

• Реєнтерабельні (reenterable) 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 (re-entrance) допускають багаторазове паралельне використання, але без їх переривання.

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 керування 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 підтримує не тільки 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 взаємно доповнюваних і взаємно діючих 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 функціювання комп'ютерної системи.

Програма (program) – дані, призначені для керування 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 – Unix, Linux, Microsoft Windows, Mac OS X, QNX;

• *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, загублених 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 встановлювати 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, несанкційованого 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 несанкційованого 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 (compilers);
- інструменти розробника (software development kit);
- системи контролю версій;
- системи відстеження 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 функціювання 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 застосунками 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 функціювання 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.

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, чистки і розмітки диску, резервного копіювання і стискання дисків).

– інтерфейсні оболонки;

– системи програмування;

– диспетчери реального часу;

– драйвери (програми керування пристроями введення-виведення).

Вбудовувані 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 вбудовуваною.

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.

Утиліти (utility, tool) – це спеціальне системне ПЗ, яке виконує ряд сервісних функцій, як з обслуговування самої ОС, так і з підготовки 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, зневадники (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 застосунком 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 команди подаються у цифрових кодах, то вони практично 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, Kotlin, ADA, Rust);

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 коди програм. *Сирцевий код* (початковий код, source code) – написаний на одній з мов програмування і складається з послідовних інструкцій. Сирцевий код (або декілька кодів) записаних на зберігання у файл називається *сирцевим модулем*.

Сирцева програма, написана на мові програмування високого рівня складається з *операторів*, *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.

Оператор (конструкція) – найменша автономна частина програми, яку можна виконати окремо. Оператори можуть бути прості або складні і містити вирази. Оператори поділяють на наступні категорії: умовні, циклу, переходів, позначки, конструкції-вирази.

Сирцева програма має бути перетворена у форму придатну для машинної обробки. Таке перетворення називається *трансляцією*. Загальна схема трансляції показана на рис. 2.

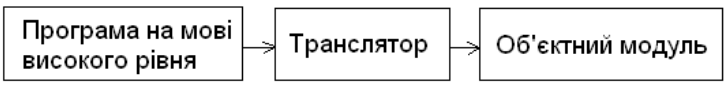

Рисунок 2 – Загальна схема трансляції

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.

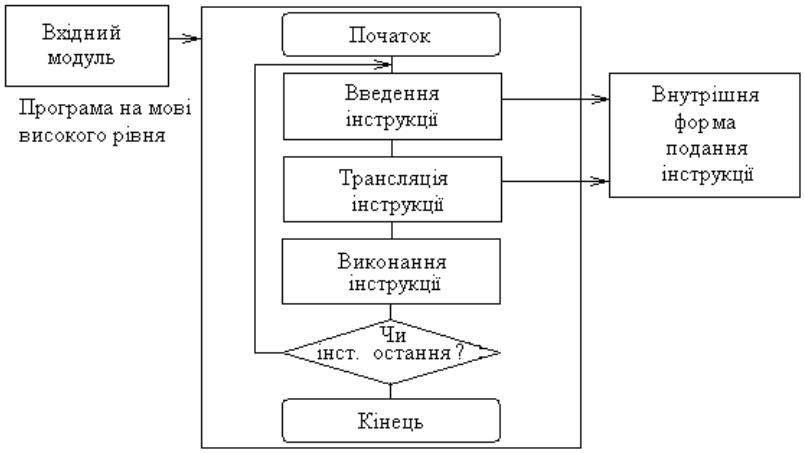

Рисунок 3 – Схема інтерпретації програми

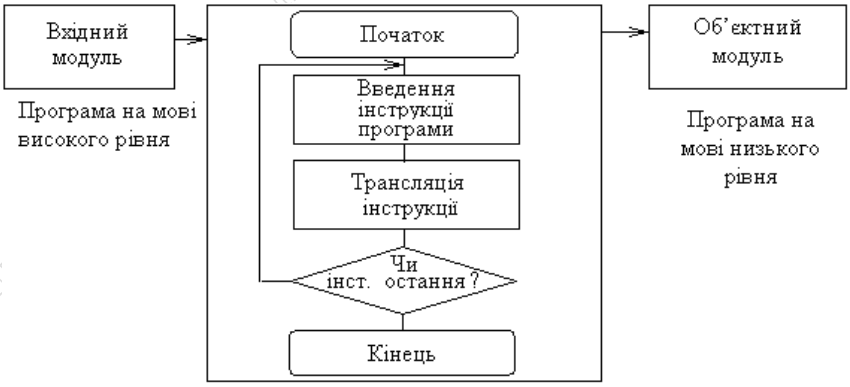

Рисунок 4 – Схема компіляції програми

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. 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.

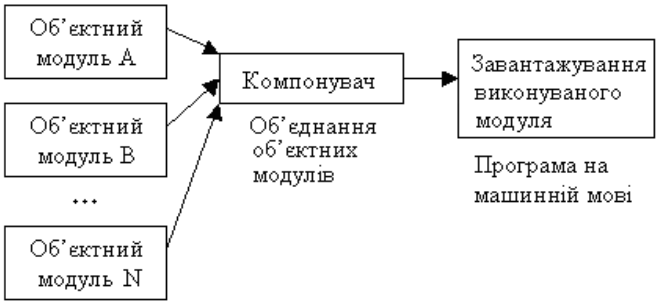

Рисунок 5 – Схема компонування виконуваного модуля

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 *переміщуваним*, тобто виконуватися на різних апаратних платформах і різних версіях ОС, як старих, так і нових. Для забезпечення переміщуваності ПЗ на системному рівні є два окремих набори визначень і описувань. Перший з них – це 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 можуть 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 задовольняють 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.

Незалежний від платформи 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 – 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 переміщуваності виконуваного коду між платформами, які мають сумісні API. На відміну від API, який регламентує сумісність на рівні сирцевого коду, AVI забезпечує сумісність на рівні *об'єктного 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 – транслятор, 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 переміщуваності 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 С.

• 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 Сі (стандарт Сі).
• 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 чистка 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 Сі `gcc` підтримує стандарт ISO 99.

6. Моделі розроблення програмного забезпечення

Найбільш відомі наступні моделі розроблення ПЗ – класична послідовна, 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.

Ітеративна модель (iterative and incremental development) передбачає розбиття ЖЦ циклу проекту на послідовність ітерацій, кожна з яких нагадує “міні-проект” з усіма фазами ЖЦ в застосуванні до створення менших фрагментів функціональності (в порівнянні з проектом в цілому).

Мета кожної ітерації – отримання працюючої версії програмної системи, що включає функціональність, визначену інтегрованим змістом усіх попередніх і поточної ітерації. Результат фінальної ітерації містить всю необхідну функціональність продукту. Таким чином, із завершенням кожної ітерації, продукт розвивається інкрементно.

З точки зору структури ЖЦ таку модель називають ітеративною (iterative). З точки зору розвитку продукту – інкрементною (incremental). Досвід індустрії показує, що неможливо розглядати кожен з цих поглядів ізольовано. Найчастіше таку змішану еволюційну модель називають просто ітеративною (говорячи про процес) та/або інкрементною (говорячи про нарощування функціональності продукту).

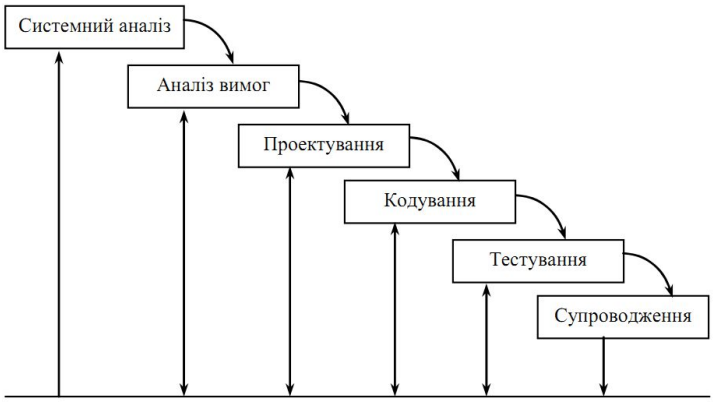

Рисунок 6 – Каскадна модель розроблення ПЗ

Еволюційна модель має на увазі не лише складання працюючої (з погляду результатів тестування) версії системи, але і її розгортання в реальних операційних умовах з аналізом відгуків користувачів для визначення змісту і планування наступної ітерації. «Чиста» інкрементна модель не передбачає розгортання проміжних збірок (релізів) системи і всі ітерації проводяться по заздалегідь визначеному плану нарощування функціональності, а користувачі (замовник) отримує лише результат фінальної ітерації як повну версію системи.

Спіральна модель (spiral model) розроблення ПЗ має вигляд серії послідовних ітерацій. На кожному витку спіралі створюється чергова версія продукту, уточняються 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.

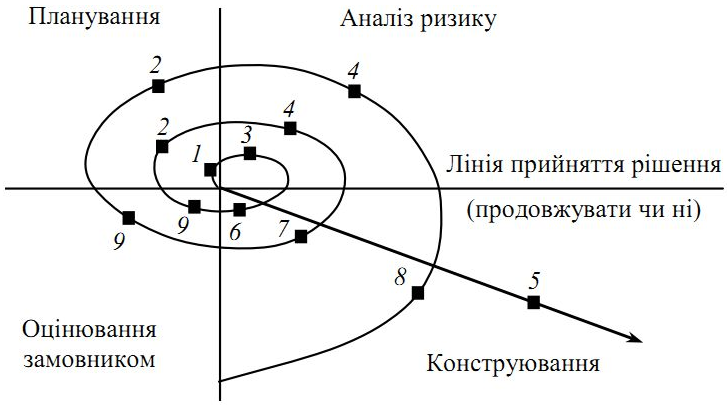

Рисунок 7 – Спіральна модель:

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).

7. Повторне використання коду

Повторне використання коду (англ. *code reuse*) – методологія проектування комп'ютерних та інших систем, яка полягає в тому, що система (комп'ютерна програма, програмний модуль) частково або повністю повинна складатися з двох частин, написаних раніше компонентів і/або частин іншої системи, і ці компоненти повинні застосовуватися більш ніж один раз (якщо не в рамках одного проекту, то хоча б різних). Повторне використання – основна методологія, яка застосовується для скорочення трудовитрат при розробці складних систем.

Відкриті програми дозволяють розробникам аналізувати *початковий* (*сирцевий*) код, знаходити можливості для його вдосконалення і використовувати його у своїх продуктах. Користувачі по всьому світу можуть вносити свій вклад в розвиток різними способами, від програмування до написання і перекладу документації, а також звітів про помилки. Вільний обмін ідеями є рушійною силою інновацій.

Згідно маніфесту GNU, люди мають моральне право на доступ до читання комп'ютерних програм.

Термін «*вільний*» має два значення: «незалежний» і «безкоштовний». Цей термін позначає ПЗ , яке можна використовувати і поширювати вільно або його можна отримати безкоштовно.

Невільне програмне забезпечення ще називається власницьким або пропрієтарним, «англ. *proprietary*». Важливо не плутати умовно-безкоштовні та безкоштовні (freeware) програми з вільними, тому шо це різні речі.

Програма вільна, якщо у її користувачів є чотири свободи:

− свобода виконувати програму в будь-яких цілях (свобода 0);

− свобода вивчати роботу програми і модифікувати програму, щоб вона виконувала потрібні обчислення (свобода 1). Це передбачає доступ до початкового коду.

− свобода передавати копії, щоб допомогти своєму ближньому (свобода 2).

– свобода передавати копії своїх змінених версій іншим (свобода 3). Цим можна дати всьому співтовариству можливість отримувати вигоду від останніх змін. Це передбачає доступ до початкового коду.

Будь-яке програмне забезпечення, користувачам якого не надається такого права, є невільним, тобто комерційним (пропрієтарним) незалежно від будь-яких інших умов.

Відкритий доступ до початкових текстів програм є ключовою ознакою вільного ПЗ, тому запропонований трохи пізніше термін open source software (ПЗ з відкритим початковим текстом) є навіть більш вдалим.

Відмінність між відкритим та вільним ПЗ полягає в основному в пріоритетах. Прихильники терміну «відкритий ресурс» роблять акцент на ефективність відкритих початкових кодів як методу розроблення, модернізації та супроводу програм. Прихильники

терміну «вільний ресурс» вважають, що саме права на вільне поширення, модифікацію і вивчення програм є головною перевагою вільного відкритого ПЗ.

Програми з відкритим початковим кодом, як правило, випускаються під вільною ліцензією.

Комітет з Програмного Забезпечення з Відкритим Початковим Кодом (*Open Source Initiative* (*OSI*)), створив на заміну терміну «вільний» термін *з відкритим початковим кодом*.

8. 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.

Ліцензії відкритого програмного забезпечення (*OSS*) повинні відповідати 10 критеріям, таким чином, щоб вважатися з відкритим вихідним кодом:

1. Вільне розповсюдження: Ліцензія не повинна обмежувати будь-яку сторону від продажу програмного забезпечення як складової частини збірки програмного забезпечення, що містить програми з декількох різних джерел. Ліцензія не повинна вимагати авторського гонорару або іншої оплати за такий продаж.

2. Початковий код: Програма повинна включати в себе початковий код і повинна допускати поширення у вигляді початкового коду, а також у компільованій формі. Там, де будь-яка форма продукту не поширюється з початковим кодом, має бути вказано засіб отримання початкового коду не більше розумної вартості відтворення, переважно завантаженням через Інтернет без пред'явлення звинувачень. Початковий код має бути у формі, в якій програміст міг би модифікувати програму. Навмисне заплутування початкового коду не допускається. Проміжні форми подання, такі як вихід препроцесора або транслятора є недопустимими.

3. Похідні роботи: Ліцензія повинна допускати модифікації і похідні роботи та має дозволити їм поширюватися на тих же умовах, що і ліцензія оригінального програмного забезпечення.

4. Цілісність авторського початкового: Ліцензія може обмежувати початковий код від поширення в модифікованій формі тільки якщо вона дозволяє розповсюдження «файлів латок» з початковим кодом для цілей модифікації файлів в процесі побудови. Ліцензія повинна явно дозволяти розповсюдження програмного забезпечення, створеного на основі модифікованого початкового коду. Ліцензія може вимагати, щоб похідні роботи мали ім'я або номер версії від оригінального програмного забезпечення.

5. Відсутність дискримінації щодо осіб або груп: Ліцензія не повинна призводити до дискримінації по відношенню до будь-якої особи або групи осіб.

6. Відсутність дискримінації щодо областей діяльності: Ліцензія не повинна обмежувати кого-небудь від використання програми в конкретній галузі діяльності. Наприклад, вона не може обмежувати програму від використання в бізнесі, або від використання в генетичних дослідженнях.

7. Поширення ліцензії: права, прикладені до програми, повинні застосовуватися до всіх, кому програма перерозподіляється без необхідності виконання додаткових ліцензій цими сторонами.

8. Ліцензія не повинна поширюватися на продукт: права на пакет не повинні залежати від програми, що є частиною конкретного пакету ПЗ. Якщо програма з пакету поширюється і використовується відповідно до умов ліцензії пакету, то всі сторони, яким надається програма повинні мати ті ж права, що і пакет.

9. Ліцензія не повинна обмежувати інше ПЗ: ліцензія не повинна накладати обмеження на інше програмне забезпечення, яке поширюється разом. Наприклад, ліцензія не повинна вказувати на те, що всі інші розповсюджувані програми, повинні бути з відкритим початковим кодом.

10. Ліцензія повинна бути технологічно нейтральною: Жодне з положень ліцензії не може ґрунтуватися на будь-якій індивідуальній технології або стилі інтерфейсу.

Самою простою і історично першою є програмна ліцензія університету Берклі – *ліцензія BSD* (Berkeley Software Distribution, 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. Але ліцензія дозволяє у майбутньому зробити продукт закритим.

Універсальна громадська *ліцензія GNU GPL* (GNU General Public License, GPL) 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.

Вносячи будь-які зміни у відкритий програмний код, розробник зобов'язується надалі надавати свої початкові коди кожному користувачеві на першу вимогу. При цьому автори знімають з себе будь-яку відповідальність за те, як буде використовуватися їх продукт і до яких наслідків може привести його використання. Єдине, що явно забороняється, − це закриття початкових кодів після їх модифікації. Принцип успадкування 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 Сі 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.

Ліцензія *MPL* (Mozilla Public License) – використовується для програм розроблених в рамках проекту Mozilla. Початковий код, скопійований або змінений під ліцензією *MPL*, повинен бути ліцензований за правилами *MPL*. На відміну від більш суворих вільних ліцензій, код під ліцензією *MPL* може бути об'єднаний в одній програмі з закритими файлами.

Тексти перерахованих вище ліцензій знаходяться у вільному доступі в мережі Internet і можуть бути вільно скопійовані і роздруковані користувачем. У тих країнах, де це не суперечить місцевому законодавству, вільні ліцензії мають чинність без підпису і печатки ліцензіата – в тому числі в електронному вигляді.

Всі перераховані ліцензії мають силу лише *англійською мовою*.

Запитання.

1. Принципи побудови системного ПЗ?
2. Базові поняття і класифікація програмного забезпечення

3. Предмет та задачі системного програмного забезпечення
4. Засоби розроблення програм.
5. Інтерфейси і стандарти
6. Яке призначення інтерфейсу прикладного програмування API?
7. Яке призначення інтерфейсу прикладного програмування AVI?
8. Моделі розроблення програмного забезпечення
9. Повторне використання коду?
10. Що таке авторське право і які є ліцензії на ПЗ?

2. ОСНОВИ ОПЕРАЦІЙНИХ СИСТЕМ

Мета. Вивчення основних понять, функцій, призначення, архітектури та можливостей 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 операційної системи
   1.2 Ресурси та їх класифікація
2. Архітектура та структура ядер ОС
   2.1. Монолітне ядро
   2.2. Мікроядро
   2.3. Екзоядро
   2.4. Наноядро
   2.5. Гібридне ядро
3. ОС Unix
4. ОС Linux
   4.1. Розміщення ядра Linux
   4.2. Модулі ядра Linux
   4.3. Компоненти дистрибутивів Linux
   4.4. Особливості дистрибутивів
   4.5. Огляд дистрибутивів
5. ОС Windows

1. Основи ОС

1.1. Поняття операційної системи

Операційна система (*ОС*) – комплекс керувальних і оброблювальних програм, які виконують завдання з 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.

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 наданий 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. 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.

За *можливістю розширення властивостей*: еластичний і нееластичний (жорсткий). 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. *Активний ресурс* здатний виконувати дії по відношенню до інших ресурсів. *Пасивний ресурс* є об'єктом на який направлені, які можуть змінити його стан або внутрішніх або. 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.

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 розглядається 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. 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. *Жорсткі 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.

За *способом переміщення*: вивантажуваний і не вивантажуваний. *Вивантажуваний ресурс* можна без наслідків забрати у володіючого ним процесу, наприклад, пам'ять. *Не вивантажуваний 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.

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.

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.

Реєнтерабельна (англ. reentrant) програмні модулі, які можуть виконуватися багаторазово без перезавантаження, причому, кожне наступне виконання може починатися до закінчення попереднього.

Крім реєнтерабельних 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.

2. Архітектура та структура ядер ОС

Ядро – це свого роду головна програма, яка є основною частиною операційної системи. Ядро виступає в ролі посередника між пристроями комп'ютера (процесором, відеокартою, оперативною пам'яттю тощо) та його програмним забезпеченням, абстрагуючи від звичайних програм і користувачів складну, низькорівневу роботу з апаратурою комп'ютера, надаючи натомість простий, зрозумілий і зручний для використання інтерфейс. Для цього в код ядра додані драйвери пристроїв, які можуть як завантажуватися в пам'ять разом з ядром ОС, так і підключатися залежно від виникнення потреби в ресурсах необхідного пристрою.

Як правило, більшість ядер ОС поділяються на три типи:

- монолітні;
- мікроядра;
- гібридні.

2.1. Монолітне ядро

Проста модель ОС з монолітним ядром, показана на рис. 1. Вона має наступну структуру:

• головна процедура, яка викликає необхідні сервісні процедури;

• набір сервісних процедур, що реалізують системні виклики;

• набір утиліт, які обслуговують сервісні процедури.

Одна сервісна процедура реалізує один системний виклик (наприклад, читати з файлу). Утиліти виконують функції, які потрібні декільком сервісним процедурам (наприклад, для зчитування й запису файлу необхідна утиліта роботи з диском).

Етапи оброблення виклику:

– приймається виклик;

– здійснюється 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.

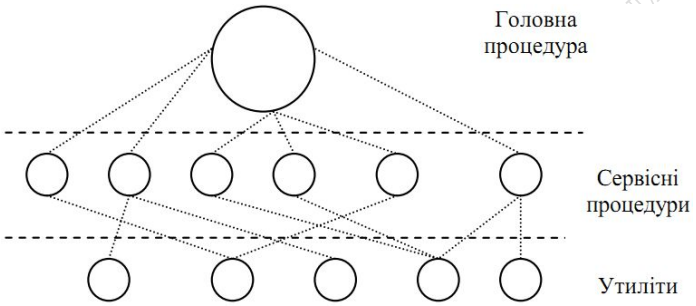

Рисунок 1 – Проста модель ОС з монолітним ядром

Реальна ОС з монолітним ядром є багаторівневою системою (рис. 2). 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.

Рівні	Функції				
7	Обробник системних викликів				
6	Файлова система 1		...		Файлова система *n*
5	Віртуальна пам’ять				
4	Драйвер 1	Драйвер 2	…	…	Драйвер *n*
3	Керування потоками				
2	Оброблення переривань, керування пам’яттю				
1	Приховування апаратури низького рівня				

Рисунок 2 – Багаторівнева ОС з монолітним ядром

Приклад багаторівневої ОС UNIX показано на рис. 3 і 4, а ОС Windows – на рис. 5.

Багаторівнево ОС з монолітним ядром має багатошарове ядро, яке виконується в привілейованому режимі. При цьому деякі допоміжні функції ОС оформлюються у вигляді застосунків і виконуються в режимі користувача. Програми користувача працюють у адресному

просторі користувача і тим самим захищені 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.

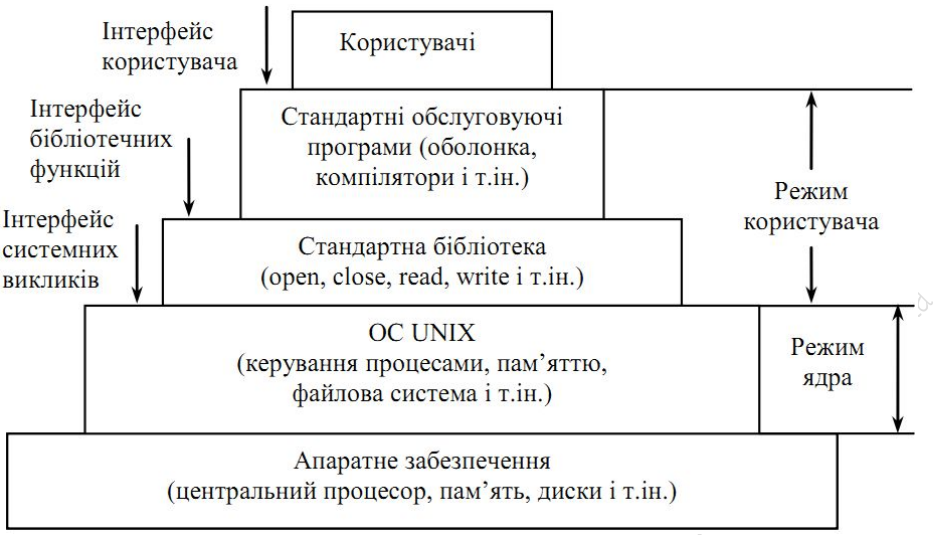

Рисунок 3 – Багаторівнева ОС UNIX

Рисунок 4 – Структура ядро ОС UNIX

Переваги і недоліки багаторівневих ОС з монолітними ядрами.

Переваги:

- практично прямий доступ програм до обладнання;
- процесам простіше взаємодіяти один з одним;
- якщо потрібний пристрій підтримується ядром, жодного додаткового встановлення ПЗ не потрібно;
- процеси реагують швидше, тому що не потрібно чекати у черзі за процесорним часом.

Недоліки:
- більший розмір ядра;
- більший розмір пам’яті, що займається;
- проблеми з безпекою, тому що всі частини працюють у просторі ядра.

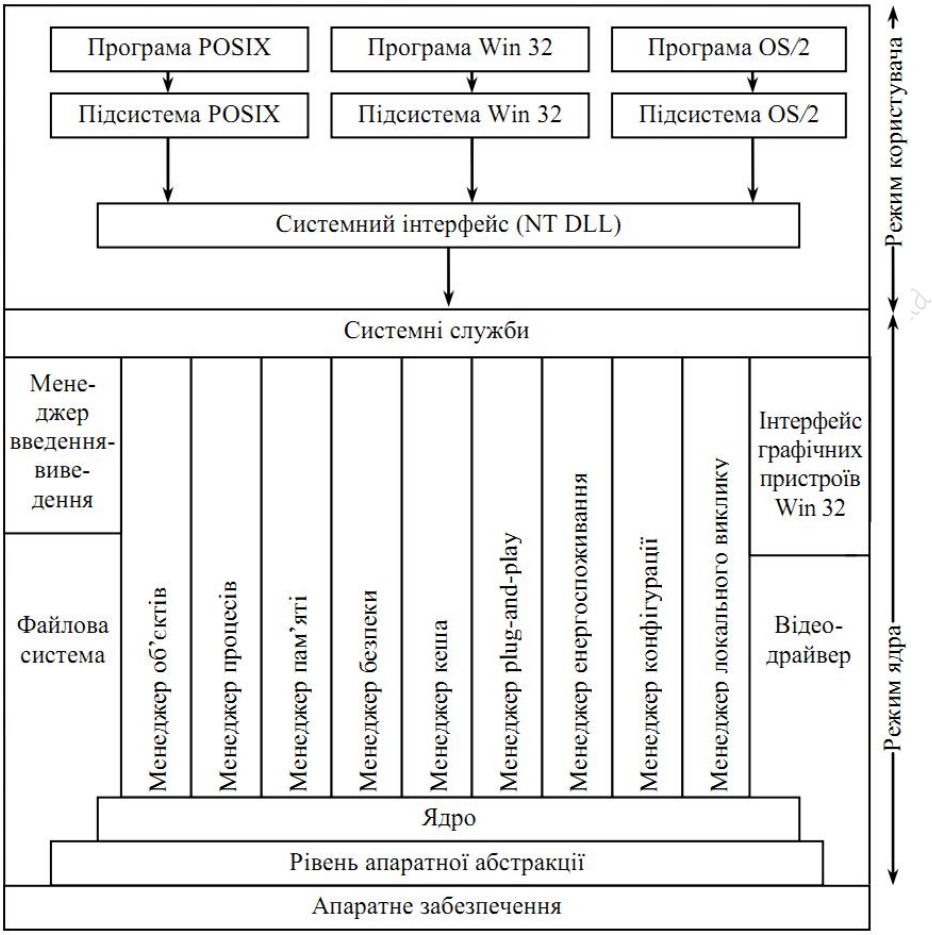

Рисунок 5 – Багаторівнева ОС Windows 2000

2.2. Мікроядро

ОС з мікроядром є альтернативою класичній багаторівневій ОС з монолітним ядром.

У мікроядрі у привілейованому режимі працює тільки дуже невелика частина ОС, рис. 6. 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, що пов’язано 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.

Усі інші більш високорівневі функції ядра оформлюються у вигляді застосунків, які працюють у режимі користувача. Однозначного рішення про те, які із системних функцій потрібно залишити в привілейованому режимі, а які перенести в користувацький режим, 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 режимі користувача.

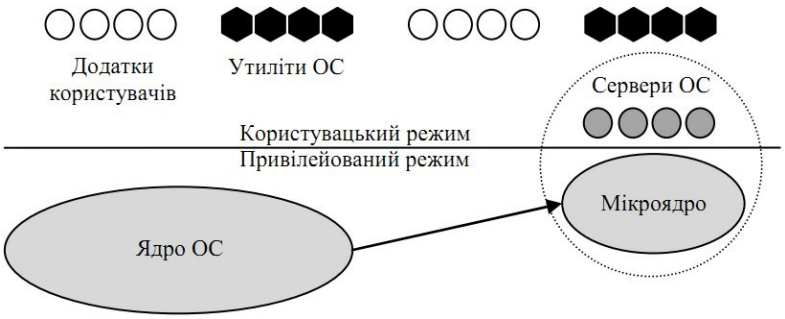

Рисунок 6 – Мікроядерна архітектура ОС

Працюючі в режимі користувача 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 іншого застосунку, виникають украй рідко. Тому в ОС із класичною архітектурою немає механізму, за допомогою якого один застосунок міг би викликати функції іншого. Коли як застосунок оформлюється частина ОС, то його основним призначенням є обслуговування запитів інших застосунків. Саме тому менеджери ресурсів, які 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. 7.

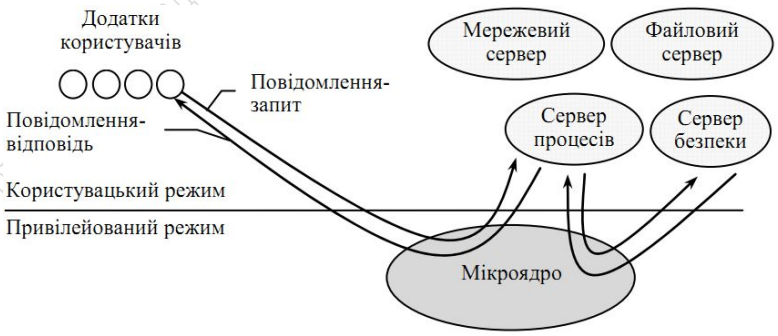

Рисунок 7 – Реалізація системного виклику в мікроядерній архітектурі

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.

Переваги й недоліки мікроядерної архітектури.

Переваги:

- переносимість;
- невеликий розмір;
-невеликі вимоги до використовуваної пам’яті;
- безпечність.

Недоліки:

- апаратне забезпечення більш абстраговане від системи;
- апаратне забезпечення може повільніше реагувати, оскільки драйвери знаходяться в просторі користувач;
- процеси не можуть отримати доступ до інших процесів без очікування.

Продуктивність. За класичної організації ОС (рис. 8, а) виконання системного виклику супроводжується двома перемиканнями режимів, а в разі мікроядерної організації (рис. 8, 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.

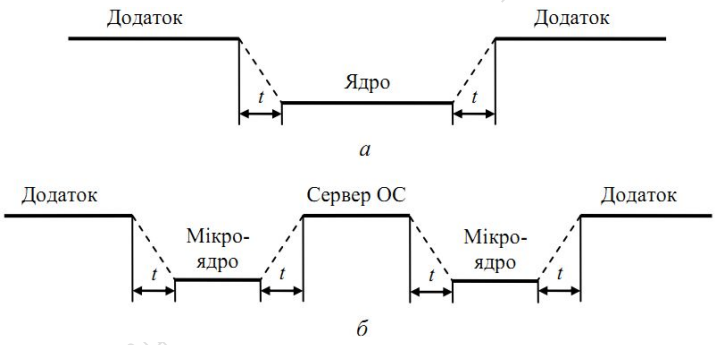

Рисунок 8 – Зміна режимів для виконання системного виклику

Головна проблема, з якою стикаються розробники мікроядерної ОС, що включати в мікроядро, а що виносити в простір користувача. 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.

2.3. 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.

2.4. Наноядро

Наноядро – архітектура ядра ОС, в рамках якої вкрай спрощене ядро виконує лише одне завдання – *оброблення апаратних переривань*, які генеруються пристроями комп'ютера. Після оброблення переривань від апаратури наноядро, у свою чергу, посилає інформацію про результати оброблення вище розміщеному 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.

2.5. Гібридне ядро

Гібридне ядро – це ядро, що поєднує у собі елементи як монолітної, так і мікроядерної архітектур. У таких ядер є можливість вибирати, які частини працюватимуть у просторі користувача (наприклад, драйвери пристроїв і система вводу-виводу файлової системи), а які – в просторі ядра (виклики міжпроцесової та серверної взаємодій). Але цей підхід має й деякі проблеми, успадковані від мікроядерної архітектури (особливо щодо швидкодії).

Переваги:

- розробник може вибирати, які програми працюватимуть в просторі користувача, а які – в просторі ядра;

– менший розмір порівняно з монолітним ядром;

– гнучкіше на відміну від інших ядер.

Недоліки:

– менша продуктивність (як і з мікроядром);

– робота драйверів пристроїв, як правило, більше залежить від виробників обладнання.

Ядро Linux *хоч і відноситься до монолітних ядер, але воно також запозичує і деякі ідеї з мікроядерної архітектури*, що означає, що вся операційна система працює в просторі ядра, а драйвери пристроїв (у вигляді модулів) можуть бути легко завантажені (або вивантажені) прямо під час роботи ОС.

3. ОС Unix

У 1969 р. Кен Томпсон у AT&T розробив ОС Unix для роботи на міні-ЕОМ на основі проекту Multics. Він пристосував цю систему, призначену, до потреб дослідників.

З часом популярність Unix і в 1970 р. Денніс Рітчі і Кен Томпсон переписали код системи мовою програмування мовою Сі. Операційна система Unix стала мобільною, тобто здатною працювати на різних типах машин без перепрограмування.

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). 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.

4. ОС Linux

У 1983 році програміст з Массачусетського технологічного інституту Річард Столмен (*Richard Stallman*), з метою створення доступної повноцінної Unix-подібної операційної системи (ОС) з відкритими початковими кодами, починає розробку **проекту GNU** (скор. від *"***G***NU is* **N***ot* **U***nix"*). Крім цього, в 1988 році з метою юридично закріпити за користувачами права на копіювання, модифікування та розповсюдження програм та початкових кодів проекту GNU, Столмен публікує **ліцензію GNU GPL** (скор. від *"***GNU G***eneral* **P***ublic* **L***icense" = "Загальна публічна ліцензія GNU"*). Однак, незважаючи на всі його успіхи, навіть через 8 років після старту проекту, не вистачало найважливішого компонента операційної системи – її ядра.

У 1991 році фіно-американським програміст Лінус Торвальдс починає роботу над власною операційною системою, взявши за основу навчальну систему ОС Minix. У 1994 року, коли, коли система стала повністю працездатною, вийшла перша основна реалізація **Linux 1.0.**

Відкриття у вільний доступ початкових кодів ОС зіграло вирішальну роль в подальшому розвитку Linux. Зауважимо, що технічно Linux – це лише ядро, без супутніх прикладних програм. Повноцінною ОС його робить супутнє програмне забезпечення. Поки що роль такого програмного забезпечення грали компоненти навчальної системи ОС Minix.

Пізніше, програми з проекту GNU замінили відповідні програми з Minix, оскільки ліцензія (GNU GPL) на початкові коди програм проекту GNU була зручнішою для використання в новій ОС.

Таким чином була створена ОС під назвою “GNU/Linux”, яку звикли називати просто “Linux”, що складається з ядра Linux, написаного Торвальдсом, і супутнього програмного забезпечення GNU, створеного в рамках проекту Столмена, рис. 9.

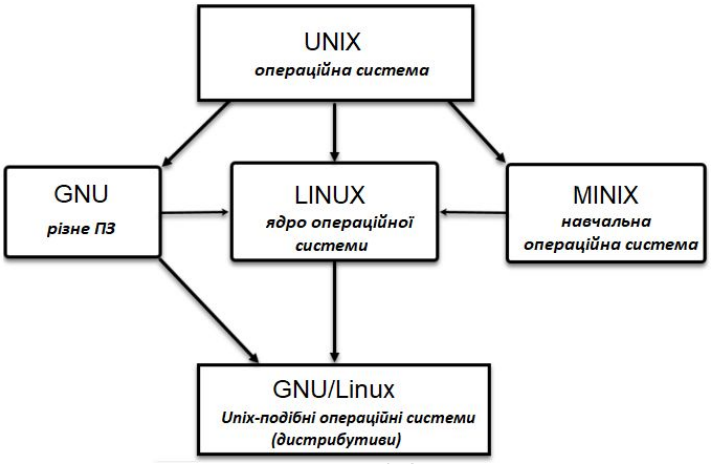

Рисунок 9 – Етапи створення ОС Linux

ОС на базі ядра Linux користуються популярністю у розробників, тому що підтримують майже всі найчастіше використовувані мови програмування: C/C++, Java, Python, Ruby, Rust тощо. Крім того, вони полегшують роботу з широким спектром корисних програм для розробки ПЗ.

ОС, що використовують ядро Linux, називаються дистрибутивами Linux, і є такими ж операційними системами як Microsoft Windows або Apple macOS, але з однією дуже важливою особливістю – їх початкові коди є відкритими, оскільки вони поширюються під ліцензією GNU GPL, яка передбачає створення вільного та відкритого програмного забезпечення (open source software). Це означає, що будь-який користувач має право вивчати та змінювати початковий код.

4.1. Розміщення ядра Linux

Щоразу під час запуску (або перезапуску) системи першим компонентом, який завантажується в пам’ять комп’ютера, є ядро Linux.

У системах Debian/Ubuntu файли присутніх в системі ядер розташовані в каталозі `/boot` і іменуються у вигляді `vmlinuz-[версія_ядра]`. Інформацію про поточну версію встановленого ядра дають команди

```
$ uname-r
vmlinuz-5.10.0--7-amd64
$ ls -l /boot
```

У каталозі `/boot` знаходяться і інші дуже важливі файли:

`initrd-[версія_ядра]` – використовується як RAM-диск, в який розпаковується і з якого завантажується ядро;

`System.map-[версія_ядра]` – використовується для керування пам'яттю до повного завантаження ядра;

`config-[версія_ядра]` – повідомляє ядру, які параметри та модулі слід завантажити в образ ядра при його компіляції.

Префікс *vm* в назві ядра є скороченням від *virtual memory*, а закінчення *z* вказує на використовуване стискання *zlib compression*.

4.2. Модулі ядра Linux

Принцип завантажуваних модулів ядра Linux (скор. "LKM" від англ. "Loadable Kernel Module") полягає в забезпеченні взаємодії ядра з усім апаратним обладнанням системи без використання при цьому всієї доступної пам'яті.

Модулі зазвичай розширюють базові можливості ядра, пов'язані з різною роботою пристроїв, файлових систем та системних викликів. Вони зазвичай мають розширення .ko і зберігаються в каталозі */lib/modules*.

Завдяки модульній структурі можна легко налаштувати ядро, встановивши необхідні модулі за допомогою *menuconfig* або відредагувавши файл */boot/config*, або можна завантажувати та вивантажувати модулі "на льоту" за допомогою команди *modprobe*.

У деяких дистрибутивах, таких як Ubuntu, доступні модулі сторонніх виробників із закритим початковим кодом. Розробники програмного забезпечення (наприклад, NVIDIA, AMD та ін.) не надають початковий код, а створюють власні модулі у вигляді попередньо скомпільованих файлів. Деякі розробники Linux вважають, що такі закриті модулі "псують" своєю присутністю ядро, так як є комерційним програмним забезпеченням, і не включають їх у свої дистрибутиви.

4.3. Компоненти дистрибутивів Linux

Ядро Linux.

Ядро Linux відповідає за зв'язок програмного та апаратного забезпечення пристрою, розподіляє системні ресурси між різними програмами, запускає процеси вводу-виводу і передає їх на виконання центральному процесору. Жодна операційна система неспроможна працювати без ядра.

На сьогоднішній день в ядрі Linux налічується понад 20 мільйонів рядків коду.

Утиліти GNU.

Утиліти GNU містять: командну оболонку bash, набір компіляторів GNU Compiler Collection, завантажувач GRUB, фреймворк GTK+, архіватор gzip, текстовий редактор Nano та інше програмне забезпечення.

GRUB – це перша програма, яка завантажується після натискання кнопки живлення комп'ютера. GRUB завантажує ядро операційної системи та інші необхідні для роботи системи компоненти. Майже 99% дистрибутивів Linux використовують завантажувач GRUB. Якщо у

комп'ютері встановлено декілька операційних систем, саме GRUB надає меню, яке дозволяє вам вибрати, яку систему слід завантажити (наприклад, Windows або Linux).

Bash – це командна оболонка. Більшість Linux-систем за замовчуванням використовують командну оболонку bash, яка надає інтерфейс командного рядка, що дозволяє керувати комп'ютером шляхом введення відповідних команд у вікні терміналу. Командні оболонки також можуть запускати сценарії, які є набором команд і операцій, що виконуються в порядку, зазначеному в тілі сценарію.

Демони (англ. "daemons") – це працюючі у фоновому режимі службові програми (або процеси), метою яких є моніторинг певних підсистем ОС та забезпечення її нормальної роботи. Наприклад, демон принтера контролює можливості друку, демон мережі контролює та підтримує мережеві комунікації тощо.

У Windows такі процеси називаються "службами", а в UNIX-подібних системах їх називають "демонами".

Менеджер пакетів – це набір програмного забезпечення, що дозволяє керувати процесом встановлення, вилучення, налаштування та оновлення різних компонентів програмного забезпечення.

У Linux програмне забезпечення поширюється пакетами. Якщо потрібно встановити програму, бібліотеку або щось інше, то для цього непотрібно шукати файл інсталяції в Інтернеті. Все, що потрібно зробити – це відкрити центр керування пакетами/програмним забезпеченням, знайти та встановити потрібні програми.

Потрібно звернути увагу на формат використовуваних пакетів. Red Hat та інші родини дистрибутивів Linux використовують формат rpm-пакетів, що мають розширення .rpm (аналогічно .exe у Windows). Системи на базі Debian Linux застосовують систему керування пакетами dpkg, яка працює з пакетами формату .deb.

У Linux існує багато різних менеджерів пакетів і вони відрізняються залежно від дистрибутиву. Наприклад, Ubuntu використовує менеджер пакетів apt, у той час як Fedora використовує dnf, openSUSE використовує zypper, а Arch Linux використовує pacman.

Наприклад, якщо потрібно у системі встановити переглядач Firefox, то слід виконати наступну команду:

Користувачам Ubuntu: sudo apt install firefox.

Користувачам Fedora: sudo dnf install firefox.

Користувачам openSUSE: sudo zypper install firefox.

Користувачам Arch Linux: sudo pacman install firefox.

Дисплейний сервер (або "**віконний інтерфейс**") – це важлива частина ОС, що відповідає за відображення на екрані графічного інтерфейсу користувача. Значки, вікна, меню, всі графічні об'єкти, які ви бачите на екрані, відображаються сервером дисплея. Без дисплейного сервера буде доступним тільки інтерфейсом командного рядка.

Існує багато різних дисплейних серверів. Для UNIX-подібних систем і дистрибутивів Linux найбільш відомим є X.Org Server, який реалізований ще 1987 році і використовується до сьогодні.

Примітка: Оскільки X.Org Server існує вже більше 30 років, він має певні проблеми безпеки. У відповідь на це деякі розробники, підтримувані такими компаніями, як Red Hat та Intel, розробили новий протокол відображення під назвою Wayland.

Оточення робочого столу (або **"середовище робочого столу"**) – це різновид графічних інтерфейсів користувача, заснований на метафорі робочого столу, що полегшує роботу з ОС за

допомогою деякого специфічного набору інструментів, а саме: значків, вікон, панелей, меню, віджетів, файлового та дисплейного менеджерів тощо. Найбільш відомими оточеннями робочого столу в Linux є GNOME та KDE.

Дисплейні менеджери використовуються для відображення екрану привітання користувача та запуску сеансів робочого столу: вони запитують ім'я користувача та пароль, перше ніж дозволити увійти в оточення робочого столу. Дисплейним менеджером GNOME є GDM, дисплейним менеджером KDE є KDM (за замовчуванням) або SDDM.

Можна вибрати будь-який дисплейний менеджер, але не допускається одночасно запускати більше одного дисплейного менеджера.

Користувацькі програми — це звичайні програми, які використовуються кожний день, наприклад: переглядач Firefox, офісний пакет LibreOffice, медіа програвач VLC тощо. Всі ці програми можуть відрізнятися в залежності від дистрибутиву.

Варто зазначити, що можна запускати будь-яку Linux-програму в будь-якому оточенні робочого столу, але програми, призначені для одних оточень робочого столу, в інших оточеннях робочого столу можуть неправильно відображатися або заважати сусіднім процесам. Наприклад, якщо спробувати запустити файловий менеджер *GNOME Files* (раніше *Nautilus*) у KDE, то він вимагатиме від встановлення різних бібліотек GNOME і, ймовірно, запуску процесів оточення робочого столу GNOME у фоновому режимі під час його відкриття. Але при цьому файловий менеджер можна використовувати.

Користувацькі програми (як і всі інші пакети) завантажуються з відповідних репозиторіїв.

4.4. Особливості дистрибутивів

Дистрибутиви Linux – це програмні пакети для встановлення операційної системи, що включають в себе ядро системи та великий набір різного програмного забезпечення.

Хоча всі дистрибутиви Linux базуються на ядрі Linux, однак у кожного з них є набір певних ознак, що характеризують той чи інший дистрибутив:

• Архітектура – тип процесорів, які підтримує дистрибутив.

• Система ініціалізації – основний підхід до запуску та управління процесами.

• Менеджер пакетів – заданий за замовчуванням інструмент управління пакетами дистрибутиву.

• Оточення робочого столу – графічний користувацький інтерфейс дистрибутиву.

Архітектура.

Архітектура – це тип процесорів, які підтримує дистрибутив. Найбільш поширені архітектури процесорів:

• x86 (або i586/i686) – 32-бітовий процесор, сумісний з Intel та AMD.

• x86_64 – 64-бітовий процесор, сумісний з Intel та AMD.

• ARM – архітектура процесора, оптимізованого для використання на мобільних пристроях (планшети, смартфони).

• PowerPC – застаріла архітектура процесорів, що застосовувалися свого часу на комп'ютерах компанії Apple.

Система ініціалізації

Система ініціалізації – це перший процес (daemon), який запускається під час завантаження комп'ютера з ОС на базі ядра Linux і функціонує протягом усього часу роботи системи. Він є батьківським процесом кожного наступного процесу, який запускається на пристрої.

Питання вибору системи ініціалізації є одним з "гаряче обговорюваних": є як прихильники/противники системи ініціалізації **SysV**, так і прихильники/противники системи **systemd**. А враховуючи, що це програмне забезпечення визначає те, яким чином система керуватиме процесами, то вибір стає не таким простим і тривіальним, як може здатися на перший погляд.

SysV – це традиційна система ініціалізації, що сягає своїм корінням до Unix System V. Вона вважається більш стабільною, але, можливо, менш функціональною, ніж systemd.

systemd – це більш сучасна система ініціалізації. Працює швидше, ніж SysV, а також дозволяє не тільки запускати процеси один раз при завантаженні, але й дає можливість стежити за ними (чи запустився процес, чи "впав" процес тощо).

Більшість сучасних дистрибутивів використовують systemd.

Менеджер пакетів

Менеджер пакетів (або «пакетний менеджер») – це заданий за замовчуванням інструмент управління пакетами дистрибутиву.

У Linux все програмне забезпечення постачається у вигляді пакетів. Роботу з архівування та управління даними пакетами виконують пакетні менеджери. Більшість пакетів не є взаємовиключними, хоча такі утиліти, як alien, можуть допомогти з конвертацією певних типів пакетів.

Різні дистрибутиви мають різні менеджери пакетів:

• **dpkg** – керує Debian-орієнтованими пакетами (.deb), поширеними в дистрибутивах на базі Debian Linux, включаючи Ubuntu та Linux Mint, за допомогою інструменту **apt** (скор. від "advanced packaging tool").

• **RPM Package Manager** – встановлює/керує пакетами формату .**rpm**. Використовує такі інструменти, як dnf, yum та zypper.

• **flatpak** – незалежний від платформи формат пісочниці/контейнера.

• **pacman** – пакетний менеджер в Arch Linux та похідних від нього дистрибутивів.

• **portage** – розроблений для Gentoo Linux, а тепер також використовується ChromeOS та декількома іншими дистрибутивами.

• **snap** – специфічна для Ubuntu форма розгортання контейнерних додатків.

• **eopkg** – використовується в дистрибутиві Solus.

Оточення робочого столу

Більшість дистрибутивів підтримують встановлення найрізноманітніших варіантів робочих столів. Кращі варіанти робочих столів дотримуються балансу між ступенем своєї конфігурованості та відносним споживанням ресурсів. Сучасні робочі столи, як правило, мають менше опцій налаштування – вони дбають більше про естетичний дизайн та зовнішній вигляд.

Конфігурація оточень робочого столу:

• До більш налаштовуваних робочих столів відносяться Xfce, LDXE, Cinnamon, MATE та KDE.

• До менш налаштовуваних робочих столів відносяться DDE (Deepin), GNOME 3 та Pantheon.

Незмінна частина дистрибутивів

У будь-якому дистрибутиві залишаються незмінними наступні принципи:

• Завжди є ядро Linux. Ядро є основним компонентом дистрибутивів Linux. Ядро – це інтерфейс між апаратним забезпеченням комп'ютера (клавіатури, миші, дисплеї тощо) та його програмним забезпеченням.

• Стандартне програмне забезпечення GNU (такі інструменти як bash, ls, rm тощо). Здебільшого це утиліти командного рядка, які становлять основну (але критично важливу) частину будь-якої Linux-системи. Можна вважати, що ядро – це автобус, який курсує між апаратним та програмним забезпеченням комп'ютера, а програмне забезпечення GNU – це набір інструментів, який потрібний для утримання автобуса на дорозі.

• Програмне забезпечення загального призначення, що постачається разом із дистрибутивом Linux. Звичайно цей список включає: текстові редактори, веб-переглядач, поштовий клієнт, (можливо) текстовий процесор або офісний пакет тощо.

4.5. Огляд дистрибутивів

Існує багато компаній та організацій, які створили свої власні дистрибутиви чи різновиди Linux-систем, які зможуть задовольнити потреби будь-якого користувача.

Debian – дистрибутив, що зібраний величезною спільнотою добровольців. Дебіан має великий вибір пакетів (близько 25 тис.) і підтримує велику кількість платформ. Дистрибутив традиційно відомий тим, що відстає від деяких інших дистрибутивів з точки зору наявності найсучасніших пакетів, але компенсує це хорошою стабільністю, оскільки основні пакети добре протестовані.

Ubuntu була випущена в 2004 році компанією Canonical. Це найвідоміший дистрибутив Linux, який є відгалуженням від Debian Linux. Ubuntu постачається з великою кількістю встановлених додатків і ще більшою кількістю найрізноманітнішого ПЗ, що перебувають у її репозиторіях. Існує багато різних збірок на основі Ubuntu:

• Edubuntu – збірка, орієнтована на навчальні заклади.

• Kubuntu – в якості оточення робочого столу використовується KDE.

• Lubuntu – полегшена версія Ubuntu.

Linux Mint

Linux Mint заснований на Ubuntu і орієнтований на користувачів-початківців, а як оточення робочого столу можна використовувати Cinnamon, Xfce, MATE, LXDE або KDE.

Red Hat Enterprise Linux

У Red Hat Linux комерційний дистрибутив із повноцінною підтримкою. Більшість користувачів RHEL застосовують його як серверну ОС, а не настільну.

CentOS

CentOS – це безкоштовна версія RHEL, яка є бінарно-сумісною з RHEL (тобто має таке саме програмне забезпечення). Багато компаній, яким не потрібна комерційна підтримка, використовують CentOS.

Fedora

Коли Red Hat перейшла на комерційну модель розповсюдження своїх продуктів, вона також випустила дистрибутив під назвою Fedora.

Fedora – це ультрасучасний, повністю безкоштовний, настільний дистрибутив Linux від Red Hat. За замовчуванням у ньому використовується оточення робочого столу GNOME, однак, як і у випадку з Ubuntu, існує безліч різноманітних збірок на основі Fedora. Оскільки дистрибутив Fedora завжди прагне бути на вістрі технологій, то його стабільність може бути нижчою порівняно з іншими дистрибутивами (Debian або Ubuntu LTS).

openSUSE

SUSE використовує як комерційну так і безкоштовну модель поширення свого дистрибутиву.

SUSE Linux Enterprise Server – це універсальний комерційний продукт, який включає службу підтримки користувачів і орієнтований на підприємства. Як утиліта конфігурації ОС та встановлення пакетів у дистрибутивах Linux від SUSE використовується YaST.

Повністю безкоштовна версія Linux від SUSE називається openSUSE. Також є версія openSUSE Tumbleweed — система, в якій постійно з'являються оновлення програмного забезпечення. У дистрибутиві є різні інструменти розробника програмного забезпечення. Утиліта openQA створена для автоматизованого тестування програмного забезпечення, тоді як Kiwi створює образи Linux для розгортання на реальному обладнанні. За замовчуванням OpenSUSE використовує оточення робочого столу KDE.

elementary OS

elementary OS – це настільний дистрибутив на базі Ubuntu. Він неймовірно інтуїтивно зрозумілий для нового користувача, що прийшов з іншої системи (особливо з macOS). Деякі з його найцікавіших функцій включають середовище робочого столу під назвою Pantheon, яке бере приклад з інтерфейсу macOS.

Gentoo

Gentoo – це вільна ОС на базі Linux. Завдяки тому, що пакети з програмним забезпеченням збираються з початкових кодів безпосередньо на комп'ютері користувача, система може бути автоматично оптимізована та налаштована практично під будь-яке апаратне забезпечення чи завдання.

Серцем Gentoo є *portage* – потужна та гнучка система налаштування та розповсюдження програмного забезпечення (менеджер пакетів), яка виконує багато ключових функцій. Наприклад, встановлюючи нове програмне забезпечення, portage автоматично створює версію користувача пакета, оптимізованого безпосередньо під цільове обладнання, гарантуючи, що в пакеті не буде непотрібного функціоналу. Завдяки portage Gentoo може стати ідеальним захищеним сервером, робочою станцією розробника, вбудованим рішенням або чимось іншим. Через його майже необмежену адаптивність, Gentoo часто називають метадистрибутивом.

Kali Linux

Kali Linux – це дистрибутив Linux на базі Debian, що містить кілька сотень програм і утиліт, націлених на вирішення різних завдань, що стосуються таких аспектів інформаційної безпеки, як:

- тестування на можливість проникнення в комп'ютери та комп'ютерні мережі;
- дослідження вразливостей веб-застосунків;
- комп'ютерна криміналістика;
- реверс-інжиніринг програмного забезпечення та багато іншого.

Як середовище робочого столу пропонуються Xfce, GNOME, KDE Plasma, LXDE або MATE. Kali Linux підтримує широкий спектр пристроїв із процесорами, побудованими на базі архітектури ARM.

5. ОС Windows

Перші версії Windows. Перша версія Windows вийшла в світ наприкінці 80-х років і залишилася абсолютно незаміченою. 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 9X. 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 була 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 постачання 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 в 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-бітового 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 версія Windows вийшла осінню 2014 року. Windows 10 може працювати на всіх сучасних гаджетах (смартфон, планшет, 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 кишенькових комп'ютерів Windows CE не є лідером, поступаючись PalmOS та іншим конкуруючим продуктам.

Windows 11.

Windows 11 – найновіша комерційна версія лінійки операційних систем (ОС) Windows NT компанії Microsoft, яка випущена 2021 року.

Системні вимоги для Windows 11 були підвищені з міркувань безпеки в порівнянні з Windows 10. Windows 11 офіційно підтримується тільки на пристроях, що використовують процесор Intel Core восьмого покоління або новіші. Хоча ОС може бути встановлена на непідтримувані процесори, не гарантується наявність оновлень. У Windows 11 вилучена підтримка 32-розрядних процесорів x86 і пристроїв, які використовують прошивки BIOS.

Windows 11 отримала змішаний резонанс на старті продажів. Перед випуском операційної системи основна увага приділялася її більш жорстким вимогам до апаратного забезпечення, обговорювалося, чи були вони законно спрямовані на підвищення безпеки Windows, чи як спосіб апсейлу клієнтів на новіші пристрої, а також питання електронних відходів, пов'язаного із цими змінами. Після випуску її хвалили за покращений візуальний дизайн, керування вікнами та більшу увагу до безпеки, але критикували за різні зміни в аспектах користувацького інтерфейсу, які вважалися гіршими, ніж у попередниці.

У 2022 році випущено велике оновлення для Windows 11 для підтримки додатків Android.

У 2023 році випущено оновлення захисника операційної системи Microsoft Defender, який є вбудованим антивірусом. Тепер він автоматично виявлятиме піратські копії операційних систем, різні програми-здирники, трояни та бекдори. Microsoft Defender може визначати піратські версії ОС. Він не лише надсилатиме повідомлення про те, що ОС “піратська”, але й скидатиме активацію та блокуватиме службу AutoKMS (Key Management Service), яку використовують зловмисники, щоб безкоштовно активувати Windows.

Наприкінці 2024 року Microsoft анонсувала нову функцію ОС, за допомогою якої можна бездротовим способом передавати файли з Android-смартфона на ПК. За допомогою нової опції, можна перейменовувати, переміщувати, вилучати та копіювати файли зі смартфона на комп’ютер, а також переносити файли з прямо з провідника Windows 11 на телефон. Також додано можливість швидко завершувати завислі програми без необхідності відкривати “Диспетчер завдань”. Ця прихована функція дозволяє завершити процес простим клацанням правої кнопки миші на іконці програми на панелі завдань.

В 2024 року, в налаштування розділу «Конфіденційність та безпека» ОС Windows 11, було додано нову функцію. Відтепер Windows 11 записуватиме всі дії користувача. Опція, яка зберігає всі дії на комп’ютері, має назву «Recall And Snapshots». Історія зберігається на спеціальній часовій шкалі, за допомогою якої можливо згадати про будь-які дії користувача, тобто всі каталоги, файли або програми, які він відкривав або запускав.

На кінець 2024 року, частка ринку Windows 11 досягла позначки в 30 %, багато в чому завдяки впровадженню нових функцій на основі ШІ, доступних тільки для ПК

Запитання

1. Дати визначення терміну ОС.
2. Дати визначення терміну операційне середовище.
3. Дати визначення терміну ресурс.
4. Які бувають архітектури ОС?
4. Які особливості монолітної ОС?
5. Які особливості мікроядерної ОС.
6. Які особливості гібридної ОС.
7. ОС Unix.
8. ОС Linux.
9. ОС Windows.

3. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РОЗРОБЛЕННЯ ПРОГРАМ

Мета. Вивчення інструментальних засобів розроблення програм.

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
   1.4. Робота в Git
2. Інтегровані середовища розроблення програм та редактори коду
3. Засіб описання компіляції та побудови програм make
   3.1. Складні командні рядки
   3.2. Змінні
   3.3. Суфіксні правила
4. Налагодження і трасування програм, виявлення помилок у пам’яті
5. Інтерпретатори і оболонки

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, компілюється 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 назві `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;

• платформонезалежна 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 – популярна система контролю версій, створена Лінусом Торвальдсом у 2005 році. Вона використовується для:

- відстеження змін у початковому коді;
- відстеження того, хто вніс зміни;
- співпраці в кодуванні.

Git забезпечує наступні функції:

- керування проектами за допомогою сховищ;
- клонування проекту для роботи з локальною копією;
- контроль та відстеження змін за допомогою Staging and Committing;
- розгалуження та злиття, щоб дозволити роботу над різними частинами та версіями проекту;
- витягнення (англ. pull) останньої версії проекту до локальної копії;
- насилання локальних оновлень в основний проект.

Робота з Git:

- Ініціалізувати Git у каталозі (`$git init`), зробивши цей каталог сховищем (англ. Repository). Git створює прихований каталог `.git` і тепер буде стежити за зміною файлів у цьому каталозі.

- Коли файл додається, змінюється, або вилучається у каталозі, він вважається зміненим (modified). Файли у Git сховищі можуть бути у двох станах: із стеженням (`untacked`) і без стеження (`tracked`). `untacked` – це файли, які є у робочому каталозі, але не добавлені у сховище. `tracked` – це файли, про які знає Git і які добавлені у сховище. Коли файли вперше додаються до порожнього сховища, усі вони не відстежуються.

Стан файлів у сховищі можна отримати командою:

```
$ git status
```

Короткі позначки стану файлів:

• ?? - невідстежувані файли;

• A - файли, додані в сцену;

• M - змінені файли;

• D - вилучені файли.

- Щоб змусити Git відстежувати файли, потрібно додати їх до проміжного середовища (Staging):

```
$ git add file1 file2
```

Замість задання імен окремих файлів можна вказати всі зміни файлів (нові, змінені, вилучені)

```
$ git add --all
```

- Зміни у файлах з проміжного середовища з додатковою інформацією можна зафіксувати у сховищі командою:

```
$ git commit -m  "Перші зміни у файлах"
```

- Для створення нової/окремої версії сховища, гілки використовується команда:

```
$ git branch project-branch-1
```

- Для переходу від основної гілки `master` до нової гілки `project-branch-1` використовується команда:

```
$ git checkout project-branch-1
```

Кожний розробник, який використовує Git, має своє локальне сховище, яке дозволяє локально керувати версіями. Потім, збереженими в локальному сховищі даними, можна обмінюватися з іншими користувачами.

Часто при роботі 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 завантажують 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:

• Git фокусується на змінах у файлі, а не на самому файлі. Коли файл додається у систему, то відслідковуються зміни файлу. При виконанні команди `git add file`, не записується файл, а позначається поточний стан файлу, фіксація змін в якому буде виконана пізніше;

• при початковому (першому) створенні репозиторію і синхронізації його з іншими розробниками, потрібний достатньо тривалий час для завантаження даних, особливо, якщо проект великий, так як потрібно скопіювати на локальний комп'ютер увесь репозиторій;

• можливість (надзвичайно мала) співпадіння хеш-коду різних за змістом версій.

1.4. Робота в Git

Порядок роботи з Git:

1. Встановлення Git клієнту:

```
$ yum install git
$ apt-get install git
$ zypper in git
```

```
git git-core git-cvs git-email git-gui gitk git-svn git-web perl-Git
(1/9) Installing: git-core-2.35.3-150300.10.15.1.x86_64
(2/9) Installing: perl-Git-2.35.3-150300.10.15.1.x86_64
(3/9) Installing: gitk-2.35.3-150300.10.15.1.x86_64
(4/9) Installing: git-gui-2.35.3-150300.10.15.1.x86_64
(5/9) Installing: git-web-2.35.3-150300.10.15.1.x86_64
(6/9) Installing: git-svn-2.35.3-150300.10.15.1.x86_64
(7/9) Installing: git-email-2.35.3-150300.10.15.1.x86_64
(8/9) Installing: git-cvs-2.35.3-150300.10.15.1.x86_64
(9/9) Installing: git-2.35.3-150300.10.15.1.x86_64
```

2. Встановлення імені користувача та адреси електронної пошти:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your_email@gmail.com"
```

3. Створення каталогу Git

```
$ mkdir /home/user/Documents/Git
```

4. Створення у каталозі Git робочого каталогу `work`

```
$ cd /home/user/Documents/Git
$ mkdir work
```

5. Створення у каталозі `work` каталогу і файлу `hello`

```
$ mkdir hello
$ cd hello
$ touch hello.html
$ cat ”Привіт світ” > hello.html
```

6. Створення репозиторію з каталогу `hello`

```
$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:   git config --global init.defaultBranch <name>
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:   git branch -m <name>
Initialized empty Git repository in /home/user/Documents/Git/work/hello/.git/
```

7. Добавлення у репозиторій сторінки `hello.html`

```
$ git add hello.html
$ git commit -m “First Commit”
[master (root-commit) 911e8c9] First Commit
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 hello.html
```

8. Перевірка поточного стану репозиторію

```
$ git status
on branch master
nothing to commit, working tree clean
```

Команда перевірки стану повідомляє, що немає проіндексованих змін (змін, які очікують на запис), які потрібно зробити (`commit`) у репозиторії). Це означає, що у репозиторії зберігається поточний стан робочого каталогу.

9. Внесення змін у файл

Змінити вміст файлу `hello.html` на:

```
<h1>Привіт світ!</h1>
```

10. Перевірка стану робочого каталогу і виявлення змін

```
$ git status
```

```
On branch master
Changes not staged for commit:
(use ”git add <file>…” to update what will be committed)
(use ”git restore <file>…” to discard changes in working directory)
      modified: hello.html

no changes added to commit (use ”git add” and/or ”git commit -a”)
```

Git виявив, що файл `hello.html` було змінено, але інформація про зміни ще не зафіксована у репозиторії. Повідомлення про стан репозиторію також підказує, що необхідно зробити. Якщо потрібно внести інформацію про виявлені зміни у репозиторій, то це можна зробити командою `git add`. Скасувати інформацію про виявлені зміни у робочому каталозі можна командою `git restore`.

11. Індексація змін (добавлення змін, які очікують на запис у репозиторій)

```
$ git add hello.html
$ git status
On branch master
Changes to be committed:
(use ”git restore –staged <file> …” to unstage)
      modified:  hello.html
```

Виявлені зміни файлу `hello.html` було проіндексовано. Це означає, що git тепер знає про зміни і вони проіндексовані, але ще не внесені у репозиторій. Відмінити індексацію змін можна командою `git restore -staged <file>`.

12. Фіксація (commit) змін у репозиторії

```
$ git commit -m “Зміни в hello.html 10/05/2023”
[master 935d387] Зміни в hello.html 10/05/2023
1 file changed, 2 insertions(+), 1 deletions (-)
```

13. Індексація і фіксація змін у репозиторії

Окремий крок індексації дозволяє вносити зміни у декількох файлах, а потім зафіксувати їх декількома командами commit. Припустимо, що відредаговано три файли (a.html, b.html, c.html). Потрібно зафіксувати зміни в a.html та b.html, а потім в у c.html.

```
$ git add a.html
$ git add b.html
$ git commit -m ”Зміни у файлах a і b”

$ git add c.html
$ git commit -m ”Зміни у файлах c”
```

14. Скасування локальних змін (до індексації)

Внести зміни у файл hello.html у вигляді небажаного коментаря:

```
<h1>Привіт світ!</h1>
<!-- Це небажаний коментар. Потрібно його вилучити. -->
```

14.1. Перевірка стану поточного каталогу

```
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hello.html
no changes added to commit (use "git add" and/or "git commit -a")
```

Видно, що файл `hello.html` було змінено, але ще не проіндексовано

14.2. Використання команди `checkout` Для перемикання у версію файлу `hello.html` у репозиторії. Виконання команд:

```
$ git checkout hello.html
Updated 1 path from the index
$ git status
On branch master
nothing to commit, working tree clean
$ cat hello.html
<h1>Привіт світ!</h1>
```

15. Скасування проіндексованих змін

Внесені зміни у файл:

```
<h1>Привіт світ!</h1>
<!-- Це небажаний коментар, але проіндексований
```

Індексування змін і перевірка статусу:

```
$ git add hello.html
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   hello.html
```

Статус показує, що зміни було проіндексовано.

Очищення буферної зони від змін:

```
$ git restore --staged hello.html
```

Повернення до індексованої версії:

```
$ git checkout hello.html
Updated 1 path from the index
$ git status
On branch master
nothing to commit, working tree clean
$ cat hello.html
<h1>Привіт світ!</h1>
```

Як видно, небажаний рядок не є більше частиною файлу.

16. Скасування зафіксованих змін (після команди commit)

Змінений файл `hello.html`:

```
<h1>Привіт світ!</h1>
<-- Це небажані, але зафіксовані зміни -->
```

```
$ git add hello.html
$ git commit -m "Ой, це небажані зміни"
[master 6fac2cf] Ой, це небажані зміни
1 file changed, 1 insertion(+), 1 deletion(-)
```

Скасування останніх зафіксованих змін:

```
$ git revert HEAD –no-edit
```

17. Перевірка журналу log показує небажані та скасовані зафіксовані зміни репозиторію:

```
$git log
```

2. Інтегровані середовища розроблення програм 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`, `geany`, `sublime`, `brackets`, `gedit`, `kate`, `eclipse`, `kwrite`, `nano`, `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.

Visual Studio (VS)– це інтегроване середовище розробки (IDE), розроблене Microsoft. Воно використовується для розробки комп'ютерних програм, включаючи веб-сайти, веб-програми, веб-сервіси та мобільні програми.

VS містить редактор коду , що підтримує IntelliSense. Інтегрований налагоджувач працює як налагоджувач на рівні початкового коду та як налагоджувач на рівні машини. Інші вбудовані інструменти включають профайлер коду, конструктор для створення програм GUI, веб-дизайнер, конструктор класів і конструктор схем бази даних. VS приймає плагіни, які розширюють функціональні можливості майже на всіх рівнях, включаючи додавання підтримки систем керування сирцевими кодами (наприклад, Subversion і Git) і додавання нових наборів інструментів, таких як редактори та візуальні дизайнери для доменно-спеціалізованих мов.

VS підтримує 36 різних мов програмування Вбудовані мови включають C/C++ , C++/CLI , Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML і CSS . Підтримка інших мов, таких як Python, Ruby, Node.js і M, доступна через плагіни. Найпростіша версія VS Community, доступна безкоштовно.

Visual Studio містить *редактор коду*, який підтримує підсвічування синтаксису та завершення коду за допомогою IntelliSense для змінних, функцій, методів, циклів і запитів LINQ.

Visual Studio містить *налагоджувач* , який працює як на рівні джерела, так і на рівні машини. Він може приєднуватися до запущених процесів, контролювати та налагоджувати ці процеси. Якщо початковий код для запущеного процесу доступний, він відображає код під час його виконання.

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 комерційний 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, C/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, (C, 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 C 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, make, mc. Завдяки такій специфіці Geany використовується при розробленні програм на різних мовах програмування, серед яких C, 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. Засіб описання компіляції та побудови програм make

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. Команда `/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.

3.1. 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.

3.2. 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
```

3.3. 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.

4. Налагодження і трасування програм, виявлення помилок в пам'яті

Для *покрокового виконання, аналізу і виявлення помилок у виконуваному файлі* використовується налагоджувач (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 ім'я_виконуваного_файлу
```

Gdb не продивляється значення PATH при пошуку виконуваного файлу. Gdb завантажує символьну інформацію для виконуваного файлу і видає запит на подальші дії.

Існує три способи перевірити запущений 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, where, w	виводить трасування стеку
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 перераховувані 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)
```


4. Інтерпретатори і оболонки

Інтерпретатори і оболонки служать різним цілям у контексті програмування та операційних систем.

Визначення: *інтерпретатор* (англ. interpreter) – це програма, яка безпосередньо виконує код, написаний на мові програмування, переводячи його в машинний код рядок за рядком або оператор за оператором під час виконання.

Функціональність: інтерпретатор обробляє мови програмування високого рівня (наприклад, Python, Ruby або JavaScript) і дозволяє розробникам запускати код без необхідності компілювати його в окремий виконуваний файл.

Визначення: *оболонка* (англ. shell) – це інтерфейс командного рядка, який дозволяє користувачам взаємодіяти з операційною системою, виконуючи команди, запускаючи програми та керуючи файлами.

Функціональність: оболонка інтерпретує команди користувача та перетворює їх на дії, які виконує операційна система. Оболонки також можуть виконувати сценарії, написані мовами сценаріїв оболонки (наприклад, Bash (Bourne Again Shell), Csh (C Shell), Zsh (Z Shell) або PowerShell).

Ключові відмінності

Призначення:

- інтерпретатор насамперед орієнтований на виконання коду з мов програмування;

- оболонка орієнтована на взаємодію з операційною системою та керування системними завданнями.

Виконання:

- інтерпретатори безпосередньо запускають код і часто надають такі функції, як налагодження та інтерактивне виконання;

- оболонки виконують команди та сценарії, дозволяючи маніпулювати файлами, контролювати процеси та виконувати завдання системного адміністрування.

Середовище:

- інтерпретатор зазвичай працює в контексті середовища програмування;

- оболонка працює в середовищі операційної системи, забезпечуючи інтерфейс командного рядка для взаємодії з користувачем.

Хоча і інтерпретатори, і оболонки інтерпретують команди або код, вони виконують різні ролі в програмуванні та взаємодії системи.

4.1. Основні можливості оболонок

Оболонка Sh

Оболонка Sh розроблена Стівом Борном в AT&T Bell Labs і вважається першою оболонкою UNIX. Sh отримав популярність завдяки своїй компактності та високій швидкості роботи. Саме це зробило його типовою оболонкою для ОС Solaris. Sh також використовується як оболонка за замовчуванням для всіх сценаріїв адміністрування системи Solaris.

Однак оболонка Sh має деякі серйозні недоліки:

- не має вбудованих функцій для обробки логічних і арифметичних операцій;

- не може відновити раніше використані команди;

- вистачає комплексних функцій для належного інтерактивного використання.

Повне ім'я шляху до оболонки Bourne - /bin/sh і /sbin/sh. За замовчуванням Sh використовує підказку # для користувача root і $ для користувачів без root.

Оболонка Bash

Стандартною оболонкою, яка постачається з більшістю систем на основі Linux, є оболонка Bash (Bourne-Again Shall).

Bash – це оболонка Unix і мова команд, написана Брайаном Фоксом для проекту GNU і вперше випущена в 1989 році. Це один із найвідоміших пакетів у колекції програмного забезпечення GNU, яку підтримує Фонд вільного програмного забезпечення (FSF). Як оболонка за замовчуванням для більшості дистрибутивів Linux, це оболонка з відкритим кодом. Вона підтримує історію команд, завершення команд для шляхів до файлів, змінних і команд. Вона також підтримує символи узагальнення, такі як *, ?, кольорові списки каталогів, виділення тексту, історію каталогу, стек або подібні функції, неявну зміну каталогу, підказку значення, індикатор прогресу та підказку параметрів. Однак вона не підтримує обов’язковий запит аргументів, автоматичні пропозиції, підсвічування синтаксису, автовиправлення, інтегроване середовище, фрагменти, контекстно-залежну довідку чи конструктор команд.

Насправді Bash також може використовувати деякі непідтримувані функції, згадані вище, але це вимагає додаткової інсталяції, а його налаштування та розширення не такі прості, як Zsh або Fish. Синтаксис команд bash є надмножиною синтаксису команд оболонки Bourne.

Оболонка Zsh

Zsh був спочатку написаний Полом Фалстадом у 1990 році в Прінстонському університеті. Zsh розроблено для інтерактивного використання, а також є потужною мовою сценаріїв. Багато корисних функцій Bash, Ksh і Tcsh були включені в Zsh. Наразі це оболонка за замовчуванням як для Mac OS, так і для Kali Linux, а також є оболонкою з відкритим кодом. Синтаксис Zsh дуже схожий на Bash, але його налаштування та розширення легші, ніж у Bash.

Оболонка Fish

Fish це розумна та зручна оболонка командного рядка для Linux, macOS. Оболонка, була випущена Акселем Лільєнкранцем у 2005 році. Fish — це скорочення від «дружньої інтерактивної оболонки» (англ. the friendly interactive shell) , і її найбільшою особливістю є зручність і простота використання. Багато інших оболонок потребують налаштування для певних функцій, але Fish надає їх за замовчуванням без необхідності будь-якого налаштування.

Fish є типовою оболонка для GhostBSD, однак підтримує менше інтерактивних функцій, ніж Zsh. Fish не підтримує обов'язковий запит аргументу, контекстно-залежну довідку, конструктор команд та індикатор прогресу

Це дуже унікальна оболонка, і її синтаксис значно відрізняється від Zsh і Bash. Це може ускладнити для користувачів Fish обмін сценаріями з користувачами Bash або Zsh. Крім того, оскільки зараз більшість систем Linux використовують Bash, використання Fish може ускладнити спільну розробку.

Оболонка Tcsh

Кен Грір, автор оболонки Csh, розширив її такими функціями, як доповнення імен файлів і редагування командного рядка і випустив її у 1983 році як Tcsh. Це типова оболонка для FreeBSD, яка підтримує менше інтерактивних функції, ніж Bash. Синтаксис такий самий, як у оболонки Csh, тому користувачі оболонки можуть легко перейти на Tcsh.

Запитання.

1. Призначення і області застосування систем контролю версій.
2. Порівняльна характеристика систем контролю версій Subversion, Mercurial, Git.
3. Команди роботи з системою контролю версій GIT.
4. Призначення і можливості інтегрованих середовищ розроблення програм.
5. Порівняльна характеристика інтегрованих середовищ Visual Studio, Eclipse, KDevelop, MonoDevelop, QtCreator, Anjuta.
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`.
9. Основні можливості оболонок ОС Linux

4. КОМАНДИ ОС LINUX

Мета. Вивчення основних груп і 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. Компоненти операційної системи Linux
   1.1. Встановлення Linux у Windows
   1.2. Команди ОС Linux і їх групи
   1.3. Команди Linux впорядковані за абеткою
2. Системна інформація
3. Файли і каталоги
   3.1. Пошук файлів
   3.2. Перегляд і редагування файлів
   3.3. Архівування і стискання файлів
4. Дисковий простір
5. Користувачі і групи
6. Встановлення програмних пакетів
   6.1. Встановлення програмних пакетів в SUSE
   6.2. Встановлення програмних пакетів в Fedora, RedHat (YUM)
   6.3. Встановлення програмних пакетів в Debian, Ubuntu (DEB)
7. Перетворення наборів символів і файлових форматів
8. Файлові системи
   8.1. Аналіз файлових систем
   8.2. Створення файлових систем
   8.3. Монтування файлових систем
9. Створення резервних копій (backup)
10. Робота з CDROM
11. Використання ресурсів і пристроїв
12. Мережа (LAN і WiFi)
13. Microsoft Windows networks (SAMBA)
14. Керування міжмережевим екраном IPTABLES (firewall)
15. Моніторинг і налагодження
16. Каталоги файлової системи Linux

1. Компоненти операційної системи Linux

Основними компонентами операційної системи Linux є:

Завантажувач: ця частина програмного забезпечення керує способом завантаження комп'ютера. У більшості випадків це сприймається користувачами як заставка, яка з'являється перед тим, як поступитися місцем, коли операційна система бере верх.

Ядро: ядро – це базовий рівень ОС. Це також частина, яку називають "Linux". Ядро є основною частиною операційної системи та керує ЦП, пам'яттю та периферійними пристроями.

Демони: фонові служби, які запускаються під час завантаження або після доступу до операційної системи.

Оболонка: Подібно до підказки ОС Windows, це командний процес, який дозволяє користувачеві керувати комп'ютером, вводячи команди через текстовий інтерфейс. Оболонка запускається і працює в терміналі.

Графічний сервер: підсистема, яка відображає графіку на моніторі.

Середовище робочого столу: частина, з якою взаємодіє користувач. На робочому столі розміщуються ярлики вбудованих програм.

Програми: назви програм, які недоступні, якщо в системі встановлено базову операційну систему. Вони доступні для завантаження в систему.

1.1. Встановлення Linux у Windows

Встановлення та налаштування Windows Subsystem for Linux (WSL) на Windows 10 або Windows 11 досить просте завдання. WSL дозволяє використовувати Linux-схожі середовища прямо в середовищі Windows без необхідності встановлення віртуальної машини або подвійного завантаження систем.

Послідовність встановлення WSL у Windows:

1. Зробити доступним WSL для Windows:

• Відкрити PowerShell як адміністратор.

• Запустити команду:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
```

• Якщо потрібний тільки WSL 1 перейти до кроку 6.

• Якщо потрібний WSL 2 перевантажити машину і перейти до кроку 2.

2. Перевірити версію Windows:

Для WSL 2 потрібний Windows 10 версії 1903 або новішої, або Windows 11.

3. Зробити доступною віртуалізацію:

Перевірити чи апаратно ввімкнена віртуалізація:

• Відкрити BIOS/UEFI на комп'ютері та увімкнути віртуалізацію.

• Увімкнути функцію Hyper-V в параметрах Windows (Панель управління -> Програми -> Увімкнути або вимкнути вимкнення Windows).

• Відкрити PowerShell як адміністратор.

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

• Перевантажити машину

4. Встановити оновлення ядра Linux для WSL 2:

```
wsl.exe --istall
```

або

```
wsl.exe --update
```

5. Встановити версію WSL 2 за замовчуванням при інсталяції нових версій Linux:

• Відкрити PowerShell і виконати команду

```
wsl --set-default-version 2
```

6. Встановити дистрибутив Linux:

Переглянути список доступних дистрибутивів Linux:

```
> wsl --list --online
Ubuntu                                    Ubuntu
Debian                                    Debian GNU/Linux
kali-linux                                Kali Linux Rolling
Ubuntu-18.04                              Ubuntu 18.04 LTS
Ubuntu-20.04                              Ubuntu 20.04 LTS
Ubuntu-22.04                              Ubuntu 22.04 LTS
OracleLinux_7_9                           Oracle Linux 7.9
OracleLinux_8_7                           Oracle Linux 8.7
OracleLinux_9_1                           Oracle Linux 9.1
openSUSE-Leap-15.5                        openSUSE Leap 15.5
SUSE-Linux-Enterprise-Server-15-SP4       SUSE Linux Enterprise Server 15 SP4
SUSE-Linux-Enterprise-15-SP5              SUSE Linux Enterprise 15 SP5
openSUSE-Tumbleweed                       openSUSE Tumbleweed
```

• Встановити потрібний дистрибутив

```
wsl --list -d <DistroName>
```

1.2. Команди ОС Linux і їх групи

Користувач взаємодіє з Linux через оболонку шляхом введення команд у терміналі. *Команда* – це назва програми, яку користувач вводить у терміналі для виконання певного завдання. Команди можуть мати додаткові дані та параметри/опції. Команди в Linux/Unix чутливі до регістру. Виконання введеної команди здійснюється лише після натискання клавіші Enter. Команди можуть виконуватися групами

```
command-1;command-2;command-3
{command-1;command-2} > test.txt
```

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           довідкова система help
help help      довідкова система help
help pwd       показати інформацію про команду pwd
man –h         довідкова система help
mount –h       показати опції команди mount
man mount      знайти man сторінки з описанням команди mount
man -k sort    знайти сторінки в базі man сторінок, які містять слово mount
man hier       виведення на екран описання ієрархії файлової системи Linux
info mount     показати інформацію про команду mount
```

```
type mount      вивести інформацію про тип команди
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:

```
whereis mount   показати binary, source і man сторінки для mount
which umount    знайти розміщення команди umount у файловій системі
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.

1.3. Команди Linux впорядковані за абеткою (470+ команд)

A

a2p – конвертує awk-сценарій у Perl-сценарій.
ac – відображає статистику про час підключення користувачів (у годинах) на основі входів/виходів із системи.
access – перевіряє, чи має програма, яка була викликана, доступ до зазначеного файлу.
accton – використовується адміністратором для увімкнення/вимкнення ведення журналу дій користувачів (log-файлу). Реєстрація активності користувачів зберігається у заданому текстовому файлі.
aclocal – автоматичне створення файлів *aclocal.m4* на основі вмісту файлів *configure.in*.
acpi – відображення стану батареї та іншої інформації підсистеми ACPI (скор. від *"Advanced Configuration and Power Interface"*).
acpi_available – перевіряє, чи доступна підсистема ACPI.
acpid – забезпечує інтелектуальне керування живленням та використовується для сповіщення програм користувацького простору про події ACPI.
addgroup – додає групу до системи.
addr2line – конвертує адреси в імена файлів та номери рядків.
adduser – додає нового користувача до системи.
agetty – Linux-версія getty, яка є Unix-програмою, що працює на головному комп'ютері та керує фізичними або віртуальними терміналами для забезпечення багатокористувацького доступу.
agrep – шукає у файлі записи, що містять рядки які точно або приблизно відповідають заданому шаблону.
alias – створення або видалення псевдоніма для команди або серії команд
amixer – консольний мікшер звукової карти, що працює під керуванням ALSA (скор. від *"Advanced Linux Sound Architecture"* — набір драйверів та утиліт для підтримки звуку в Linux).
aplay – консольний аудіоплеєр для звукових карт, що працюють під керуванням ALSA.
aplaymidi – використовується для відтворення стандартних MIDI-файлів (скор. від *"Musical* **I***nstrument* **D***igital* **I***nterface"*) шляхом надсилання вмісту MIDI-файлу на MIDI-порт ALSA.
apropos – пошук команди за ключовим словом, пов'язаним з командою. Показує короткий опис команд, у яких є шукане слово.
apt – потужна консольна система керування пакетами, за допомогою якої відбувається встановлення, оновлення та видалення програмного забезпечення в операційних системах на основі Debian Linux.
apt-get – консольна утиліта, яка допомагає в керуванні та обробці пакетів у Linux.
aptitude – багатофункціональний менеджер пакетів у Linux, що є оболонкою для apt.
ar – використовується для створення, зміни та вилучення файлів з архівів.
arch – відображає інформацію про архітектуру комп'ютера.
arp – керування системним ARP-кешем (скор. від "Address Resolution Protocol"). Також дозволяє створити його повний дамп.
as – асемблер проекту GNU; використовується компілятором GCC.

aspell – використовується як засіб для перевірки орфографії в Linux.
at – виконує команди у заданий час.
atd – демон планувальника завдань, що запускає завдання, поставлені в чергу за допомогою команди at.
atrm – видалення вказаних завдань, доданих через команду at. Щоб видалити вибране завдання, необхідно передати команді його номер.
a**tq** – відображає список відкладених завдань, запланованих користувачем.
autoconf – створення конфігураційних скриптів для пакетів з вихідним кодом.
autoheader – створення файлу шаблону операторів #define або будь-якого іншого заголовка шаблону.
automake – автоматичне створення makefile-ів (файлів, що відповідають стандартам кодування GNU).
autoreconf – створення вихідного коду, що автоматично збирається, для Unix-подібних систем.
autoupdate – оновлення файлу configure.in у системі Linux до нової версії Autoconf.
awk (gawk) – потоковий редактор з вбудованою мовою програмування для керування даними та створення звітів

B

banner – відображення великими літерами рядка ASCII-символів у стандартний вивід.
basename – видаляє інформацію про каталог та суфікси з імен файлів, тобто виводить ім'я файлу з вилученням усіх компонентів каталогу.
base32 – кодування/декодування даних і друк в стандартний потік виведення
base64 – кодування/декодування даних і друк в стандартний потік виведення
bash – запуск командної оболонки bash (скор. від "Bourne Again Shell"). Ця оболонка є стандартною у більшості Linux-дистрибутивів.
batch – використовується для зчитування команд зі стандартного терміналу або вказаного файлу та їх виконання на дозволеному рівні навантаження системи, тобто, коли середнє навантаження падає нижче 1.5.
bc – консольний калькулятор.
bdiff – як і команда diff, застосовується для пошуку відмінностей, але у великих файлах.
bg – використовується для переміщення на передній план завдання, що виконується у фоновому режимі.
biff – система поштових повідомлень для Unix, яка сповіщає користувача у командному рядку про появу нових листів.
bind – вбудована команда оболонки bash, яка використовується для встановлення прив'язок клавіш та змінних Readline.
bison – генератор синтаксичного аналізатора, схожий на yacc.
bmon – монітор пропускної здатності та оцінювач швидкості в мережі
break – використовується для завершення виконання циклів for, while та until.
builtin – виконує вбудовану команду оболонки. Використовується тоді, коли ви хочете замінити вбудовану команду оболонки деякою функцією, але при цьому вам потрібна функціональність вбудованої команди всередині самої функції.
bye – аналог команди exit, використовується для завершення сесії або виходу з термінала.
bzcmp – виклик утиліти cmp для стиснених файлів формату .bzip.

bzdiff – порівняння стиснених файлів формату .bzip2.
bzgrep – пошук шаблону або виразу, але всередині стисненого файлу формату .bzip2.
bzip2 – блоко-орієнтований стискач/розтискач файлів.
bzless – схожа на команду bzmore, але має набагато більше функцій. bzless не потрібно читати весь вхідний файл перед запуском, тому з великим файлом вона запускається швидше, ніж текстові редактори (наприклад, vi).
bzmore – це фільтр, який дозволяє переглядати як стислі (.bzip2), так і прості текстові файли.

C

cal – перегляд календаря певного місяця або цілого року. За замовчуванням показує календар поточного місяця.
calendar – служба нагадувань.
caller – повертає вміст любого виклику активної підпрограми
cancel – зупинення виводу інформації про стан виконання завдання.
case – альтернатива кільком операторам if/elif (коли використовується одна змінна).
cat – зчитує дані з файлу та відображає їх вміст як вихідних даних.
cc – використовується для компіляції коду мови Сі та створення виконуваних файлів.
ccrypt – інструмент командного рядка для шифрування та дешифрування даних.
cd – зміна поточного робочого каталогу.
cfdisk – утиліта для перегляду та керування таблицею розділів диску.
chage – утиліта для перегляду та зміни інформації про закінчення терміну дії пароля користувача.
chattr – зміна атрибутів файлу в каталозі.
chdir – зміна робочого каталогу (аналог команди cd).
checkeq – процесор мови програмування для опису рівнянь та виконання порівнянь.
checknr – перевірка nroff-і troff-файлів на помилки.
chfn – дозволяє легко змінити ім’я користувача та інші деталі.
chgrp – зміна групи, яка володіє файлом або каталогом.
chkconfig – відображення поточної інформації про запуск служб або будь-якої конкретної служби, а також оновлення налаштувань рівня запуску служби.
chmod – використовується для керування дозволами на заданий файл/каталог.
chown – зміна власника файлу чи групи.
chpasswd – зміна пароля одночасно для декількох користувачів.
chroot – зміна кореневого каталогу.
chrt – керування атрибутами реального часу процесу.
chsh – зміна оболонки входу користувача (поточної оболонки).
chvt – перемикання між різними доступними терміналами TTY (скор. від “Teletypewriter“).
cksum – відображення та обчислення значення контрольної суми файлу або CRC (скор. від “Cyclic Redundancy Check”), його розміру в байтах та імені у стандартному виводі у терміналі.
clear – очищення екрану термінала.
cmp – побайтове порівняння двох файлів. Допомагає з’ясувати, чи ідентичні два порівнювані файли.
col – фільтрує переходи рядків із вхідного потоку.

colcrt – форматування виводу текстового процесора таким чином, щоб його можна було переглядати на дисплеях з ЕПТ (скор. від “Електронно-Променева Трубка”).
colrm – вилучає вибрані стовпці з рядків файлу. Стовпець визначається як один символ у рядку. Вхідні дані зчитуються зі стандартного входу. Результат записується у стандартний вивід.
column – форматування відображення вмісту файлу у вигляді стовпців.
comm – порівнює два відсортовані файли або потоки по рядках та записує у стандартний вивід: спільні та унікальні рядки.
command - виконати *command* з *аргументами* ігноруючи любі функції оболонки з іменем command
compgen - виведення списку усіх команд Linux
compress – зменшення розміру файлу. Після стиснення файл набуває розширення .Z.
continue – пропуск поточної ітерації в циклах for, while та until.
cp – копіювання файлів або каталогу.
cpio (скор. від “copyin, copy out”) – обробка архівних файлів, таких як *.cpio або *.tar. Ця команда може копіювати файли до архівів та з архівів.
cron – утиліта, що автоматизує виконання запланованого завдання у вказаний час.
crontab – список команд, які потрібно виконувати по розкладу, а також ім’я команди, яка використовується для керування цим списком.
csh – командна оболонка C Shell.
csplit – розділення будь-якого файлу на контекстно визначені частини.
ctags – дозволяє швидко отримати доступ до файлів (наприклад, швидко побачити визначення функції).
cu – посилання сигналу через термінал іншій системі.
cupsd – планувальник підсистеми виведення CUPS (скор. від “Common Unit Printing System”).
curl – утиліта для передачі даних на сервер або із сервера з використанням будь-якого із підтримуваних протоколів.
cut – розбиття файлу на частини і виведення у стандартний вивід вибраних байтів, символів або полів з кожного рядка файлу.
cvs – збереження історії змін файлу. Щоразу, коли файл пошкоджено або щось іде не так, команда cvs допомагає повернутися до попередньої версії та відновити файл.

D

date – відображення та встановлення системної дати та часу.
dc – обчислення арифметичних виразів.
dd – утиліта командного рядка для Unix та Unix-подібних операційних систем, основною метою якої є конвертування та копіювання файлів, запис дискових заголовків, boot записів.
ddrescue – відновлення даних із зіпсованих розділів диску
declare – оголошення змінних та функцій, встановлення атрибутів та відображення їх значень.
delgroup – вилучення групи із системи.
deluser – вилучення користувача з системи.
depmod – формування списку залежностей модулів ядра та генерація відповідних *.map-файлів.
deroff – вилучає з файлів nroff- та troff-конструкції.
df – виведення інформації про доступний та використаний дисковий простір.

dhclient – утиліта для роботи з DHCP-протоколом (отримання динамічної IP-адреси, налаштування мережевих інтерфейсів та ін.).
dig – виведення інформації про DNS (скор. від “Domain Name System”).
diff – виведення відмінностей у двох файлах шляхом їх рядкового порівняння.
diff3 – виведення відмінностей у трьох файлах шляхом їх рядкового порівняння..
dig – перегляд DNS
dir – перерахування вмісту каталогу.
dircolor – задання кольору для ls
dircmp – порівняння вмісту двох каталогів.
dirname – перетворення *pathname* у *path*
dirs – виведення списку запам'ятовних каталогів
disable – деактивує принтери, відключаючи їх від запитів на друк, що відправляються командою lp.
dmesg – виводить повідомлення ядра під час початкового завантаження Linux або налаштовує їх буфер.
dmidecode – дозволяє отримати інформацію про апаратні компоненти системи, а також іншу корисну інформацію: характеристики процесора, оперативної пам’яті (DIMM), деталі BIOS тощо.
domainname – дозволяє вивести або встановити NIS/YP-доменне ім’я.
dosfsck – діагностує файлову систему MS-DOS на наявність проблем і намагається їх усунути.
dpkg – менеджер пакетів для систем на базі Debian Linux.
dpost – конвертування файлів з формату troff у PostScript.
dstat – зазвичай використовується системними адміністраторами для отримання інформації про мережеві з’єднання, пристрої вводу-виводу, процесор тощо.
du – відстеження файлів та каталогів, які займають надмірну кількість місця на жорсткому диску.
dump – резервне копіювання файлової системи на будь-який запам’ятовувальний пристрій.
dumpe2fs – дамп інформації файлової системи ext2/ext3.
dumpkeys – відображає інформацію про поточну розкладку клавіатури.

E

echo – виведення тексту/рядка на екрані, які передаються як аргумент.
ed – запуск рядкового текстового редактора з мінімальним інтерфейсом.
edit – текстовий редактор (різновид редактора для простих користувачів).
egrep – обробляє шаблон як розширений регулярний вираз і виводить рядки, що відповідають шаблону.
eject – дозволяє виймати знімний носій (зазвичай CD-ROM, дискету, стрічку, JAZ- або ZIP-диск) за допомогою програмного забезпечення.
elif – використовується для визначення оператора else if.
elm – інтерактивна поштова система.
emacs – редактор з простим інтерфейсом користувача, в якому немає режиму вставки. Має лише режим редагування.
emerge – пакетний менеджер дистрибутиву Gentoo Linux.
enable – увімкнення/вимкнення lp-принтерів.

env – виведення інформації про змінні середовища. Також використовується для запуску утиліти або команди в середовищі користувача.
eqn – використовується для опису порівнянь.
eval – команда сприймає передані їй аргументи як директиви оболонки.
ex – текстовий редактор у Linux, який також називається лінійним режимом редактора vi.
exec – використовується для виконання команди з самого bash.
exit – закриває командну оболонку зі станом N. Якщо N не вказано, станом виходу буде стан останньої виконаної команди.
expand – дозволяє конвертувати табуляції в пробіли у файлі, а коли файл не вказано, дані зчитуються зі стандартного вводу.
expect – команда, яка працює зі сценаріями, які очікують на ввід даних від користувача. Автоматизує завдання, надаючи вхідні дані.
export – позначає змінні середовища, які експортуються в дочірні процеси.
expr – обчислює заданий вираз і відображає результат.

F

factor – вивід простих множників заданих чисел (що задаються як через командний рядок, так і через стандартний ввід).
false – нічого не виконує, повертає не нульовий код повернення
fc – використовується для перерахування, редагування або повторного виконання команд, раніше введених в інтерактивну оболонку.
fc-cache – сканує каталоги шрифтів (і створює їх кеш), які використовують fontconfig для обробки шрифтів.
fc-list – використовується для перерахування доступних шрифтів та стилів шрифтів. Список всіх шрифтів можна відфільтрувати та відсортувати, використовуючи відповідну опцію форматування.
fdisk – діалогова команда в Linux, яка використовується для створення та керування таблицею розділів диска.
fg – переміщення фонового завдання на передній план.
fgrep – пошук рядків у файлі.
file – виведення типу файлу.
find – пошук файлів та каталогів.
findsmb – список всіх пристроїв, доступних за SMB-протоколом (скор. від “Server Message Block”).
finger – докладна інформація про всіх користувачів, що увійшли до системи.
fmt – утиліта простого переформатування тексту у рядки заданої ширини.
fold – обгортає кожен рядок у вхідний файл, щоб відповідати вказаній ширині, та виводить її на стандартний вивід.
foreach – виконує набір команд для кожного з елементів заданого масиву.
for – використовується для багаторазового виконання набору команд для кожного елемента, що є у списку.
format – форматує диск
free – відображення обсягу вільної пам’яті, що використовується в системі.
fsck – перевірка та відновлення файлової системи.

ftp – інтерактивна утиліта для доступу до FTP (скор. від “File Transfer Protocol”).
fun – малювання в терміналі візерунків різного типу.
function – створення функцій або методів.
fuser – визначення процесів, які використовують файли або сокети.

G

g++ використовується для попередньої обробки, компіляції, збирання та компонування вихідного коду при створенні виконуваного файлу.
gawk – GNU-версія awk.
gcc – використовується для компіляції програм, написаних мовами C, C++, Objective-C та Objective-C++.
gdb – потужний налагоджувач для програм, написаних на C, C++, Ada, Fortran та ін.
getent – отримання елементів з бази даних.
getfacl – отримання списків контролю доступу до файлів.
getopts – розбір позиційних параметрів
gpasswd – адміністрування файлів /etc/group та /etc/shadow.
gprof – виведення даних щодо профілювання програми.
grep – пошук у файлі певного шаблону символів та виведення всіх рядків, що містять цей шаблон.
groupadd – створення нової групи.
groupdel – вилучення наявної групи.
groupmod – модифікація або зміна наявної групи.
groups – відображення списку груп, до яких долучено користувача.
grpck – перевірка цілісності інформації про групи, а саме: всі записи в /etc/group та /etc/gshadow мають правильний формат і містять допустимі дані.
grpconv – перетворює паролі користувачів та груп в захищену форму.
gs – команда викликає Ghostscript — інтерпретатор мови Adobe Systems PostScript та формату PDF (скор. від “Portable Document Format”).
gunzip – стиснення або розтиснення файлів.
gview – версія редактора vi з графічним інтерфейсом. Запускається у новому вікні.
gvim – синонім для команди gview.
gzexe – стиснення виконуваних файлів, а також їх автоматичне розтиснення у момент виконання.
gzip – стиснення файлів (кожний файл в окремий архів).

H

halt – вказівка апаратному забезпеченню комп’ютера зупинити всі процеси, які виконуються в поточний момент. Основне застосування – перезавантаження або вимкнення системи.
hash – доступ до хеш-таблиці нещодавно виконаних програм.
hdparm – отримання інформації про жорсткий диск, зміна інтервалів запису, налаштувань DMA (скор. від “Direct Memory Access”).
head – виводить N перших рядків файлу.
help – виводить інформацію про вбудовані команди оболонки.

hexdump – фільтр, який відображає вказані файли або стандартний ввід, якщо файли не вказані, у заданому користувачем форматі.
history – відображає історію команд, які були введені з початку сесії.
host – утиліта для роботи з DNS-запитами.
hostid – відображає числовий ідентифікатор поточного хоста.
hostname – відображає або встановлює ім'я комп'ютера.
hostnamectl – може використовуватися для запиту та зміни імені хоста системи та пов'язаних з ним параметрів.
htop – консольна утиліта, яка дозволяє користувачеві інтерактивно (в режимі реального часу) відстежувати список запущених процесів.
hwclock – запит та встановлення апаратного годинника (скор. "RTC" від "Real-time clock").

I

iconv – перетворення вказаного тексту з одного кодування в інше.
id – відображення ідентифікатора власника, члена групи до якої входить власник, групи.
if – виконання команд за заданих умов.
ifconfig – конфігурація мережевого інтерфейсу.
ifdown – вимикає мережевий інтерфейс.
iftop – інструмент аналізу мережі, який використовується системними адміністраторами для перегляду статистики, пов'язаної з пропускною спроможністю каналу передачі даних.
ifquery – дозволяє вибрати інформацію про мережевий інтерфейс.
ifstat – дозволяє читати та контролювати різні статистичні дані мережевого інтерфейсу
ifup – використовує (підіймає) мережевий інтерфейс, дозволяючи йому передавати та отримувати дані.
import – створення скріншоту екрана (всього екрана або тільки його частини) з подальшим збереженням у файл.
info – дозволяє читати документацію у форматі info.
insmod – програма для активації модулів ядра.
install – копіює файли та встановлює атрибути.
iostat – моніторинг статистики I/O-операцій для пристроїв та розділів.
iotop – відображення статистики роботи процесів з дисками.
ip – відображення та керування мережевими підключеннями, маршрутизацією та ін.
ipcrm – видалення IPC-ресурсів (скор. від "Inter-Process Communication") та пов'язаних з ними даних.
ipcs – виведення інформації про IPC-ресурси.
iptables – налаштування правил брандмауера, що входить до складу ядра Linux.
iptables-save – зберігає поточні правила iptables у вказаному файлі.
iwconfig – відображення параметрів та статистики бездротового зв'язку, які беруться з /proc/net/wireless.

J

jobs – відображення стану завдань у поточній сесії.
join – утиліта для з'єднання рядків двох файлів на основі ключового поля, що є в обох файлах.

journalctl – використовується для перегляду логів, зібраних `systemd`. `systemd` “збирає” логи у бінарному форматі. Щоб їх подивитися, використовується команда `sudo journalctl`.

K

kill – посилає процесу сигнал завершення роботи.
killall – посилає сигнал kill всім активним процесам.
ksh – командна оболонка Korn Shell.

L

last – виведення списку всіх користувачів, що увійшли та вийшли з системи з моменту створення файлу /var/log/wtmp.
ld – редактор посилань на бібліотеки для об’єктів.
ldd – відображення залежностей спільно використовуваних бібліотек.
less – екранне читання вмісту текстового файлу.
let – обчислення арифметичних виразів для змінних оболонки.
link – створення жорсткого посилання на файл.
ln – створення символьних посилань між файлами.
lo – завершує роботу з командною оболонкою.
local – створення змінної у функції
locate – пошук файлів за іменем.
login – вхід до системи.
logname – відображення login користувача.
logout – аналог lo.
look – виводить рядки, що починаються із заданого підрядка.
losetup – налаштування та керування віртуальними loop-пристроями.
ls – відображення списком вмісту каталогу.
lsblk – відображення відомостей про блокові пристрої.
lshw – генерація докладної інформації про апаратну конфігурацію системи на основі файлів із каталогу /proc.
lsmod – відображення модулів ядра Linux, які в поточний момент завантажені.
lsof – відображає інформацію про те, які файли використовуються тим чи іншим процесом.
lspci – виводить список pci пристроїв
lsusb – відображення інформації про USB-шини та пристрої, під’єднані до них.
lzcat – переглянути вміст файлу, стисненого LZMA (скор. від “Lempel-Ziv-Markov chain-Algorithm”).
lzma – стиснути або розтиснути файл за алгоритмом LZMA.

M

mach – відображення інформації про тип процесора.
mailq – список поштових повідомлень, поставлених у чергу для подальшого відправлення.
mailx – інтерактивна система обробки повідомлень електронної пошти.

make – визначення, які частини великої програми повинні бути перекомпільовані, і відображення команд для їх перекомпіляції.
man – відображення посібника користувача по будь-якій команді, яку можна запустити в терміналі.
mapfile – читання рядків із стандартного входу у індексований масив
md5sum – перевірка цілісності даних за допомогою алгоритму хешування MD5.
mkdir – створення нового каталогу
mkfifo – створення FIFO
mkfile – створення одного або декількох файлів для NFS-монтованої swap області.
mknode – створення дискового блоку
mktemp – створення тимчасового файлу
merge – злиття вмісту трьох файлів.
mesg – надсилання повідомлень в інший термінал.
mkdir – створення одного або кількох каталогів.
mkfs – створення файлової системи у вибраному розділі.
mkswap – створення файлу (або розділу) підкачування (swap).
modinfo – відображення інформації про модуль ядра Linux.
modeprobe – програма для завантаження та вивантаження модулів з ядра Linux.
more – екранний перегляд текстових файлів у командному рядку.
most – перегляд (сторінковий) тестового файлу
mount – приєднання зовнішніх пристроїв до файлової системи.
mpstat – створення статистичного звіту про роботу процесора (або по кожному процесору, якщо їх декілька).
mt – керування роботою накопичувача з магнітною стрічкою.
mtr – діагностика мережі (traceroute/ping)
mv – переміщення всередині файлової системи одного або декількох файлів, або каталогів з одного місця в інше.
mmv – масове переміщення, копіювання, додавання файлів
mysql – програма для керування базою даних MySQL.
mysqldump – утиліта для створення резервної копії бази даних MySQL.

N

nc – потужна утиліта, яка використовується для вирішення різних завдань, пов'язаних з мережевими TCP- та UDP-з'єднаннями.
netstat – відображає різноманітну інформацію, пов'язану з мережею (наприклад, мережеві з'єднання, таблиці маршрутизації, статистика інтерфейсу тощо).
newgrp – надає користувачеві права нової групи на певний час.
nice – запуск програми зі зміненим пріоритетом.
niscat – відображення таблиць та об'єктів NIS+.
nischmod – зміна прав доступу до об'єкта NIS+.
nischown – зміна власника об'єкта NIS+.
nischttl – зміна значення часу життя об'єкта NIS+.
nisdefaults – відображення заданих за замовчуванням значень NIS+.
nistbladm – команда адміністрування таблиці NIS+.

nft – пакет фільтрування і класифікації пакетів мережі (NAT)
nl – виводить кількість рядків у файлі.
nmap – інструмент дослідження мережі та сканер безпеки/портів.
nmcli – керування NetworkManager-ом. Також може застосовуватися для відображення стану мережевого пристрою, створення, редагування, активації/деактивації та видалення мережевих підключень.
nohup – продовжити виконання команди, коли сесія термінала буде завершена.
nroff – додаток для системи форматування документів.
nslookup – інструмент адміністрування мережі, пов'язаний з DNS.

O

od – виведення файлів у вісімковому, шістнадцятковому або інших форматах.
op – оператор доступу для системного адміністратора.
open – відкриття файлів.
on – виконання команди на віддаленій системі, але з локальним оточенням.
onintr – відображення інформації про апаратні переривання.

P

pack – стиснення файлів за алгоритмом Хоффмана.
pacman – менеджер пакетів у Arch Linux.
pagesize – відображення розміру сторінки в пам'яті.
parted – програма для розмітки диска.
partprobe – інформування операційної системи про зміни в таблиці розділів.
passwd – зміна паролів облікових записів користувачів.
paste – з'єднання файлів (паралельним злиттям) шляхом виведення результуючих рядків, які складаються з рядків кожного вказаного файлу, розділених табуляцією, у стандартний вивід.
pax – читання та запис файлових архівів та копіювання ієрархій каталогів.
pact – виведення вмісту стисненого текстового файлу.
perl – інтерпретатор Perl-скриптів.
pg – фільтр для посторінкового перегляду вмісту текстових файлів.
pgrep – список процесів за іменами
pico – простий текстовий редактор у стилі Pine Composer.
pidof – визначення ідентифікаторів процесів конкретної запущеної програми.
pine – програма для Інтернет-новин та електронної пошти.
ping – перевірка мережевого підключення між хостом та сервером/іншим хостом.
pkill – завершити процес по імені.
pinky – команда пошуку інформації про користувача, яка надає докладну інформацію про всіх користувачів, що увійшли до системи. На відміну від finger, застосовуючи pinky, ви можете обрізати цікаву для вас інформацію.
pmap – відображення інформації про адресний простір процесу.
popd – відновити попереднє значення поточного каталогу
poweroff – вимкнення системи.
pr – розбиття текстових файлів на сторінки для виводу.

printenv – відображення змінних середовища.
printf – відображення відформатованих даних (числа, рядки тощо).
ps – відображення інформації про активні процеси.
pstree – відображення дерева процесів.
pushd – зберегти і змінити поточний каталог
pv – відображення переміщення даних через PIPE
pvs – відображає форматований вивід інформації про фізичні томи.
pwd – відображає повний шлях до поточного робочого каталогу.

Q

quota – показати використання диску
quotacheck – сканування файлової системи для використання диском
quit – завершення сеансу командної оболонки.

R

ram – ram диск пристрій
ranlib – генерує індекс до вмісту архіву і зберігає його в архіві.
rar – архівація файлів із стисканням
rcp – копіювання файлів з одного комп'ютера на інший.
read – зчитування рядка зі стандартного вводу.
readarray – читання з stdin у змінну масив
readelf – отримання інформації про файли формату ELF (скор. від "Executableand Linkable Format").
readlink – відображення значення символьного посилання.
reboot – перезавантаження системи.
red – запуск редактора ed у режимі прокручування тексту.
rename – перейменування файлів.
renice – зміна пріоритету процесу
repeat – повторювати виконання команди вказану кількість разів.
replace – утиліта заміни рядків у файлах.
reset – ініціалізація термінала. Корисно в тих ситуаціях, коли програма, що завершилася, залишила термінал у «ненормальному» стані.
restore – відновлення файлів із резервної копії, створеної за допомогою dump.
return – вихід із функції.
rev – реверсивна зміна рядків файлу.
rlogin – віддалений вхід до системи.
rm – вилучення файлів, каталогів, символьних посилань та ін.
rmdir – вилучення каталогів.
rmmod – вилучення модуля з ядра.
route – використовується для роботи з IP-адресами та таблицею маршрутизації.
rpcinfo – відображає інформацію про RPC (скор. від "Remote Procedure Call").
rsh – віддалений командна оболонка.

rsync – дозволяє синхронізувати файли та каталоги між двома розташуваннями. Поводиться майже так само, як rcp, але має набагато більше опцій і використовує протокол віддаленого оновлення для значного прискорення передачі файлів при оновленні цільового файлу.

S

s2p – конвертер sed-скриптів у Perl.
sar – моніторинг ресурсів системи Linux, таких як: завантаження процесора, використання пам'яті, навантаження на пристрої вводу-виводу тощо.
scp – безпечне копіювання файлів між серверами.
screen – забезпечує можливість запуску та використання кількох сеансів оболонки з однієї ssh-сесії.
screen – консольний менеджер віртуальних терміналів.
script – використовується для запису всього, що відбувається в терміналі.
scriptreplay – відтворення активності термінала, збереженої за допомогою команди script.
sdiff – знаходить відмінності між двома файлами та об'єднує їх в інтерактивному режимі.
sed – потоковий редактор для фільтрації та перетворення тексту.
select – створення нумерованого меню, з якого користувач може вибрати потрібний параметр.
sendmail – надсилання електронного листа.
seq – генерує послідовність чисел із заданим кроком.
service – утиліта для запуску та керування службами.
sestatus – перевірка того, чи включений SELinux (модуль безпеки Linux).
set – встановлення значення змінної середовища.
setfacl – налаштування списків контролю доступу до файлів.
setsid – запуск програми в новому сеансі.
sfdisk – програма для розмітки дисків.
sftp – клієнт для передачі файлів за протоколом SFTP.
sh – командна оболонка Bourne Shell.
shift – зсув/переміщення аргументів командного рядка на одну позицію вліво.
shopt – опції оболонки
showkey – відображає код кожної кнопки, яка натискається на клавіатурі.
shred – повне видалення файлу з жорсткого диска.
shuf - генерація випадкових перестановок
shutdown – безпечне вимкнення системи Linux.
sleep – затримка на вказаний час.
slocate – пошук файлів
slogin – псевдонім ssh-клієнта, який застосовується для віддаленого входу в систему через ssh.
smbclient – ftp-подібний клієнт для доступу до ресурсів SMB/CIFS.
source – читання та виконання вмісту файлу (зазвичай набору команд), що передається як аргумент у поточному скрипті оболонки.
sort – сортування рядків у текстових файлах.
spell – перевірка орфографії.
split – поділ великих файлів на дрібніші.
startx – ініціалізація сесії X-сервера.
ss – перегляд інформації про підключення до мережі.

ssh – протокол, який використовується для безпечного підключення до віддаленого сервера/системи.
stat – відображення статистики файлу або файлової системи.
stop – зупинення виконання фонового завдання.
strace – один з найпотужніших інструментів моніторингу та діагностики, що дозволяє проводити трасування системних викликів та сигналів.
strftime – форматування рядка з датою та часом.
strip – дозволяє видалити налагоджувальну інформацію з виконуваних файлів.
stty – відображає та встановлює параметри термінала.
su – перехід в оболонку із правами суперкористувача (вводиться пароль *суперкористувача*)
sudo – виконання одної команди із правами суперкористувача (вводиться пароль *користувача*)
sum – обчислення контрольної суми та підрахунку блоків у файлі.
suspend – зупинка виконання оболонки
swapoff – вимикає файл (або розділ) підкачування.
swapon – використовує файл (або розділ) підкачування
sync – синхронізація даних на диску з даними в пам'яті.
systemctl – використовується для перевірки та контролю стану systemd та управління службами.

T

tabs – зупиняє роботу вкладок у терміналі.
tac – об'єднує та виводить файли у зворотному порядку.
tail – виводить N останніх рядків файлу.
talk – програма візуальної комунікації, яка копіює рядки з вашого термінала до термінала іншого користувача.
tar – використовується для створення архіву та вилучення архівованих файлів.
tcopy – копіювання магнітної стрічки.
tcpdump – консольний аналізатор мережевого трафіку.
tcsh – командна оболонка TENEX C Shell.
tee – зчитує дані зі стандартного пристрою вводу та записує їх на стандартний пристрій виводу або у файл.
telnet – використовується для зв'язку з іншим хостом за протоколом TELNET (скор. від "Teletype Network").
test – перевіряє вирази умов.
time – виконує команду та по її завершенню виводить статистику про кількість витрачених ресурсів на її виконання.
timeout – виконання команди з часовим лімітом
times – виведення часу користувача і системи
timex – час виконання команди (з відображенням детальної інформації).
tmux – термінальний мультиплексор
todos – конвертування текстових файлів Unix у формат DOS.
top – відображення та оновлення інформації про топ процесів CPU.
touch – використовується для створення, зміни та модифікації тимчасових міток файлу.
tput – встановлення термінально залежних характеристик, кольору, позиції
tr – перетворення (або вилучення) символів заданих шаблоном у великі/малі букви

tracepath – відображає маршрут мережевими вузлами з MTU (скор. від “Maximum Transmission Unit”).
traceroute – відображає маршрут, яким пакет досягає хосту.
trap – виконання команд при отриманні оболонкою сигналів
tree – список вмісту каталогів у деревоподібному форматі.
true – нічого не виконується, успіх
tty – друк імені терміналу у stdin.
tsort – топологічне сортування
type – використовується для опису того, як інтерпретуватиметься кожен аргумент, якщо він буде використовуватися як ім’я команди.

U

ulimit – обмеження ресурсів користувача
umask – встановлює маску прав для режиму створення файлу.
umount – дозволяє розмонтувати файлову систему.
uname – відображає інформацію про систему.
unalias – видалення псевдоніму.
uncompress – розтиснення стисненого файлу.
unexpand – конвертує кожний пробіл в табуляцію, записуючи результат у стандартний вивід.
uniq – вилучає або повідомляє про рядки, що повторюються у стандартному вході.
units – перетворення з одного масштабу у інший
unix2dos – конвертує текстовий файл Unix в формат DOS.
unlink – вилучення посилання на файл.
unlzma – вилучення файлів з архіву, стисненого командою lzma.
unpack – вилучення файлів з архіву, стисненого командою pack.
unset – вилучення змінної або імені функції
until – виконання заданого набору команд доти, доки умова циклу оцінюється як помилкова.
unxz – розтиснення .xz-архіву.
unzip – розтиснення .zip-архіву.
uptime – відображає час того, як довго система працює з моменту увімкнення.
useradd – додати облікові записи користувачів до вашої системи.
userdel – вилучення облікового запису користувача та пов’язаних з ним файлів.
usermod – зміна властивостей користувача в Linux через командний рядок.
username – отримання імені користувача.
users – відображення імен користувачів, що увійшли до системи на поточний момент.

V

vacation – автовідповідач для електронної пошти.
vi – текстовий редактор.
vim – вільний текстовий редактор, створений на основі старішого vi.
vmstat – команда моніторингу продуктивності системи, яка надає інформацію про процеси, пам’ять, файли підкачування, активність процесора тощо.

vnstat – використовується системними адміністраторами для моніторингу параметрів мережі: завантаженість каналів, вхідний/вихідний трафік та ін.

W

w – відображає користувачів, що увійшли до системи та їх процеси.
wait – очікувати завершення процесу
wall – відображає у терміналі повідомлення для всіх користувачів, що увійшли у систему.
watch – періодичний запуск та стеження за програмою, відображаючи її вивід в терміналі.
wc – визначення кількості рядків, слів, байтів та символів у файлах.
wget – утиліта для завантаження файлів з Інтернету.
whatis – отримання однорядкового опису шуканої команди.
whereis – відображає повний шлях до виконуваного файлу програми (або до вихідних файлів, якщо вони присутні у системі).
which – пошук шляху $path для заданого файлу.
while – багаторазове виконання набору команд доти, доки умова в while є істинною.
who – отримання інформації про поточного користувача.
whoami – відображає ім'я поточного користувача.
whois – відображення доступної інформації про Інтернет-ресурс (наприклад, про сайт).
write – дозволяє користувачеві спілкуватися з іншими користувачами, копіюючи рядки з одного терміналу в інші.

X

xargs – перетворює вхідні дані, отримані від стандартного вводу, в аргументи команди.
xdg-open – використовується для відкриття файлу або URL-адреси у кращому для користувача додатку.
xfd – відображення всіх символів шрифту X-сервера.
xhost – налаштування прав доступу до X-сервера.
xlsfonts – відображення всіх шрифтів X-сервера.
Xorg – виконуваний файл X-сервера.
xrdb – управління базою даних ресурсів X-сервера.
xset – зміна значення змінної X-сервера.
xxd – зробити hexdump
xz – стиснення файлу в .xz-форматі.
xzcat – перегляд вмісту текстового файлу, стисненого командою xz.

Y

yacc – стандартний генератор синтаксичних аналізаторів (парсерів) в Unix-системах.
yes – виводить аргументи командного рядка із символом "\n" доти, доки команді не буде надіслано сигнал kill.
yppasswd – зміна паролю бази даних NIS.
yum – менеджер пакетів у дистрибутивах на основі Red Hat Linux.

Z

zcat – відображення вмісту файлу, стисненого командою zip, gzip.
zdiff – викликає diff для файлів, стиснутих командою gzip.
zdump – отримання інформації про часовий пояс.
zgrep – пошук виразів у заданому файлі, навіть якщо він стиснутий.
zip – стиснути файли в архів.
zipcloack – зашифрувати .zip-файл.
zipinfo – вивести інформацію про .zip-файл.
zipnote – перегляд та зміна коментарів .zip-файлів.
zipsplit – об'єднання декількох .zip-файлів.
zypper – пакетний менеджер в openSUSE.
!! – повторення введеної команди
– коментар

2. Системна інформація

arch або **uname** -m – відобразити архітектуру комп'ютера
cal -3 – календар попереднього, поточного і наступного місяця
cal 2020 – вивести таблицю-календар 2020-го року
cat /etc/SuSE-release – інформація про версію ОС Suse
cat /etc/os-release – інформація про версію ОС
cat /proc/cpuinfo – відобразити інформацію про процесор
cat /proc/interrupts – показати переривання
cat /proc/meminfo – перевірити використання пам'яті
cat /proc/mounts – відобразити змонтовані файлові системи
cat /proc/net/dev – показати мережеві інтерфейси і їх статистику
cat /proc/swaps – показати файл(и) підкачування
cat /proc/version – вивести версію ядра
clear – очищення екрану термінала
clock -w – зберегти системний час у BIOS
date 041217002007.00* – встановити системну дату і час ММДДГГxxРРРР.СС (МісяцьДеньГодиниХвилиниРік.Секунди)
date – вивести системну дату
dmidecode -q – показати апаратні системні компоненти — (SMBIOS / DMI)
exit – завершити сеанс поточного користувача
fc –s номер – виконати команду за номером
finger, **users**, **who**, **w** – інформація про користувачів системи
hdparm -i /dev/hda – вивести характеристики жорсткого диска
hdparm -tT /dev/sda – протестувати продуктивність читання даних з жорсткого диска
history !номер – виконати команду за номером
history | tail -10 – показати останні 10 введені команди
history – історія команд
!! – виконати останню команду
!n – виконати команду з номером n
last reboot – статистика останніх перевантажень
logout – вийти із системи

lsmod – список модулів завантажених у ядро
lspci -tv – показати як дерево PCI пристрої
lsusb -tv – показати як дерева USB пристрої
passwd – змінити пароль поточного користувача
poweroff – вихід з Linux
reboot – перевантаження системи
shutdown -c – відмінити заплановану за розкладом зупинку системи
shutdown -h hours:minutes & – запланувати запинку системи у вказаний час
shutdown -h now – вихід з Linux
shutdown -h now або init 0 або telinit 0 — зупинити систему
shutdown -r now або reboot – перевантажити систему
uname -r – відобразити версію ядра ОС
uptime – поточний час та робота системи без перевантаження і виключення
whois linux.org – інформація про домен linux.org
winecfg – налаштування Wine (не емулятор WinAPI)

3. Файли і каталоги

cat file – виведення вмісту файлу
cat file.txt | pv –s `du –sb file.txt | cut –f1` виведення файлу на екран з візуалізацією прогресу виконання в %.
cat > newfile – створення порожнього файлу (Ctrl+D – кінець введення)
> /home/newfile – створення порожнього файлу Ctrl+D – кінець введення)
cd $HOME – перейти в home каталог
cd $OLDPWD – перейти в попередній робочий каталог
cd - – перейти в каталог, в якому знаходилися до переходу у поточний каталог
cd .. – перейти в каталог рівнем вище
cd ../.. – перейти в каталог двома рівнями вище
cd /dev; ls –al sda* ttyS* – виведення списку файлів пристроїв
cd /home – перейти в каталог '/home'
cd /usr/bin – перейти в каталог usr/bin з root каталогу
cd – перейти в домашній каталог користувача /home/user
cd ~ – перейти в в домашній каталог користувача /home/user
cd ~user – перейти в домашній каталог користувача user
cmp file1, file2 – порівняння файлів
cp -a /tmp/dir1 . – скопіювати каталог dir1 із усім вмістом у поточний каталог
cp -a dir1 dir2 – копіювати каталог dir1 у каталог dir2
cp -la /dir1 /dir2 – копіювати каталог dir1 в каталог dir2
cp /home/file1 /home/file2 – копіювання файлу
cp /home/user1/my.txt /home/new.txt – копіювати файл /home/user1/my.txt у файл home/new.txt
cp dir/* . – копіювати всі файли каталогу dir в поточний каталог
cp file1 file2 – скопіювати файл file1 у файл file2
dd if=/dev/cdrom of=whatever.iso – копіювання ISO образа із CD або DVD
dd if=/dev/hda of=mymbrfile bs=512 count=1 – копіювання MBR із завантажувального сектора IDE жорсткого диска

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)
diff file.old, file.new > dif.txt – виведення розходжень файлів у файл
diff file.old, file.new – порівняння файлів і виведення їх розходжень
du -sh /home/Documents – визначення розміру каталогу
du -sh /home/file – визначення розміру файлу
echo "Останній рядок" | sudo tee -a /home/text – добавлення текстового рядка "Останній рядок" у кінець файлу /home/text
echo ”Команда для роботи з каналом” | pv –qL 5 – посимвольне виведення стрічки із швидкістю 5 байт/сек
echo ”Останній рядок” | sudo tee –a /home/file – добавити повідомлення в останній рядок файлу file
file – виведення типу файлу
find / -name e100 -print 2> /dev/null – пошук файлу e100, починаючи з root каталогу і направлення повідомлень про помилки в нульовий пристрій
find file – пошук файлу в ієрархії каталогів
grep – друк рядків файлу, які співпадають з шаблоном
gzip file – створення архіву файлу
head /var/log/messages – вивести початок файлу
less file – по екранне виведення файлу з рухом назад
ln -s file1 lnk1* – створити «м’яке» (символьне) посилання на файл або каталог (на шлях)
ln /user/file1 /user/file2 – створити жорсткий зв’язок між файлами (hard link) (аналог до cp –l /user/file1 /user/file2)
ln file1 lnk1 – створити «жорстке» (фізичне) посилання на файл чи каталог (на індексний номер)
ln –s /user/file1 /user/file2 – створити м’який зв’язок між файлами (soft link) (аналог до cp –s /user/file1 /user/file2)
locate file – пошук усіх файлів з іменем file
ls *[0-9]* – показати файли і каталоги, які містять у імені цифри
ls -F – відобразити вміст поточного каталогу з добавленням до імен символів, які характеризують тип
ls -a – показати скриті файли і каталоги в поточному каталозі
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 / – список каталогів файлової системи
ls – виведення впорядкованого списку каталогів і файлів
ls -laX – виведення відсортованого за розширеннями списку усіх каталогів і файлів
ls -las – виведення відсортованого за іменами списку усіх каталогів і файлів
md5sum openSUSE-10.3-RC1-KDE-i386.iso – перевірка контрольної суми файлу
mkdir -p /tmp/dir1/dir2 – створити дерево каталогів
mkdir /home/user1/newdir – створення нового каталогу /home/user1/newdir
mkdir dir1 dir2 – створити два каталоги одночасно
mkdir dir1 – створити каталог з іменем 'dir1'
mkfifo – створення іменованого каналу (черга типу FIFO)

more file – по екранне виведення файлу з рухом вперед
mv /dir1 /dir2 – перейменувати каталог dir1 в каталог dir2
mv /home/file1 /home/file2 – перейменування файлів
mv dir1 new_dir – перейменувати чи перемістити файл або каталог
nl – нумерація рядків файлу
od -vt x1 /tmp/mynullfile – перегляд вісімкового дампа файлу
pwd – показати поточний каталог
rm -f file1 – вилучити файл з іменем 'file1'
rm -rf /home/user1/dir – вилучити каталог dir з вкладеними каталогами
rm -rf dir1 dir2 – вилучити два каталоги і рекурсивно їх вміст
rm -rf dir1 – вилучити каталог з іменем 'dir1' і рекурсивно увесь його вміст
rm /home/file – вилучення файлу
rmdir /home/dir – вилучити каталог dir
rmdir dir1 – вилучити каталог з іменем 'dir1'
sha1sum openSUSE-10.3-RC1-KDE-i386.iso – перевірка контрольної суми файлу
sort – сортування рядків файлу
stat – виведення параметрів файлу, прав і часу доступу до файлу
tail /var/log/messages – вивести кінець файлу (для великих файлів)
tar –cf arx.tar *.* – створення tar-архіву усіх файлів поточного каталогу
tar –cf arx.tar file1, file2, file3 – створення tar-архіву заданих файлів
tar –cvf arx.tar subdir – створення tar-архіву підкаталогу
touch $$hello – створити порожній файл з унікальним іменем
touch -t 0712250000 fileditest – модифікувати дату і час створення файлу, а при його відсутності, створити файл з вказаними датою і часом (YYMMDDhhmm)
touch /home/newfile – створення порожнього файлу /home/newfile
touch file – створити порожній файл
touch hello.$$ – створити порожній файл з унікальним розширенням
tree або **lstree** – показати дерево файлів і каталогів, починаючи з кореня (/)
vim, **kwrite** – редагування файлів з використання редакторів
wget --convert-links -r http://www.linux.org/ – копіювати сайт повністю (на 5 рівнів в глибину) і конвертувати посилання для автономної роботи
wget http://itshaman.ru/images/logo_white.png – завантажити файл logo_white.png у поточний каталог
wget -r -k -l 7 -p -E -nc -w 1 http://site.com – завантажити сайт повністю

-r – рекурсивно переходити за посиланнями, щоб завантажувати сторінки

-l – визначає максимальну глибину вкладеності сторінок, які wget має завантажити (за замовчуванням 5, в прикладі встановлено 7)

-k – перетворювати всі посилання у скачаних файлах таким чином, щоб за ними можна було переходити на локальному комп'ютері (в автономному режимі)

-p – завантажувати всі файли, які потрібні для відображення сторінок (зображення, CSS)

-E – добавляти до завантажених файлів розширення .html

-nc – існуючі файли не перезаписувати. Це дозволить продовжити завантаження сайту, перерване в попередній раз

-w – очікувати 1 сек між завантаженнями

zip, **bzip2** – створення архівів файлів

3.1. Пошук файлів

find / -name *.rpm -exec chmod 755 '{}' \; – знайти всі файли і каталоги, імена яких закінчуються на '.rpm', та змінити права доступу до них
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 / -xdev -name "*.rpm" – знайти всі файли і каталоги, імена яких закінчуються на '.rpm', ігноруючи знімні носії, такі як cdrom, floppy і т.п.
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
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'

3.2. Перегляд і редагування файлів

cat -n file1 – перенумерувати рядки при виведенні вмісту файлу
cat example.txt | awk 'NR%2==1' – при виведенні вмісту файлу, не виводити парні рядки
cat file_originale | [operation: sed, grep, awk, grep і т.п.] > result.txt – загальний синтаксис виконання дій по обробленні вмісту файлу і виведення результату в новий файл
cat file_originale | [operazione: sed, grep, awk, grep і 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
cut – запис у стандартний вивід вибраних байтів, символів або полів з кожного рядка файлу
comm -3 file1 file2 – порівняти вміст двох файлів, вилучаючи рядки, які зустрічаються в обох файлах
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'
echo 'esempio' | tr '[:lower:]' '[:upper:]' – перетворити символи з нижнього регістра у верхній
echo a b c | awk '{print $1,$3}' – вивести перший і третій стовпці Розділення стовпців, за замовчуванням, за пропуском/пропусками або символом/символами табуляції
echo a b c | awk '{print $1}' – вивести перший стовпець. Розділення стовпців, за замовчуванням, за пропуском/пропусками або символом/символами табуляції

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
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
grep Aug /var/log/messages – із файлу '/var/log/messages' відобразити і вивести на стандартний пристрій виведення стрічки, яка містить «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 /var/log/messages – із файлу '/var/log/messages' вибрати і вивести на стандартний пристрій виведення стрічки, яка починається з «Aug»
paste -d '+' file1 file2 – об'єднати вміст file1 і file2 як таблицю з розділювачем «+»
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
sed '/ *#/d; /^$/d' example.txt – вилучити порожні рядки і коментарі з файлу example.txt
sed '/^$/d' example.txt – вилучити порожні рядки із файлу example.txt
sed 's/stringa1/stringa2/g' example.txt – у файлі example.txt замінити «string1» на «string2», результат вивести на стандартний пристрій виведення
sed -e '1d' result.txt – вилучити перший рядок з файлу example.txt
sed -e 's/ *$//' example.txt – вилучити порожні символи у кінці кожного рядка
sed -e 's/0*/0/g' example.txt – замінити послідовність з любої кількості нулів одним нулем
sed -e 's/string1//g' example.txt – вилучити стрічку «string1» з тексту не змінюючи всього іншого
sed -n '/string1/p' – відобразити тільки рядки, які містять «string1»
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
sort file1 file2 | uniq -d – відсортувати вміст двох файлів, відображуючи тільки рядки, які повторюються
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 – відсортувати вміст двох файлів, не відображуючи повторень
sort file1 file2 – відсортувати вміст двох файлів

3.3. Архівування і стискання файлів

bunzip2 file1.bz2 – розтиснути файл 'file1.bz2'
bzip2 myfile – стиснути файл з найбільш можливим ступенем стиску
gunzip file1.gz – розтиснути файл 'file1.gz'
gzip -9 file1 – максимально стиснути файл file1
gzip file1 або bzip2 file1 – стиснути файл 'file1'
lzop –d myfile.lzo – розтиснути файл
lzop –v myfile – стиснути файл з найбільшою швидкістю
rar a file1.rar file1 file2 dir1 – створити rar-архів 'file1.rar' і включити в нього file1, file2 і dir1
rar a file1.rar test_file – створити rar-архів 'file1.rar' і включити в нього файл test_file
rar x file1.rar – розпакувати rar-архів

tar -cvf archive.tar file1 file2 dir1 – створити tar-архів archive.tar, який містить файли file1, file2 і каталог dir1
tar -cvf archive.tar file1 – створити tar-архів archive.tar, який містить файл file1
tar -cvfj archive.tar.bz2 dir1 – створити архів і стиснути його за допомогою bzip2
tar -cvfz archive.tar.gz dir1 – створити архів і стиснути його за допомогою gzip
tar -tf archive.tar – показати вміст архіву
tar -xvf archive.tar -C /tmp – розпакувати архів в /tmp
tar -xvf archive.tar – розпакувати архів
tar -xvfj archive.tar.bz2 – розтиснути архів і розпакувати його
tar -xvfz archive.tar.gz – розтиснути архів і розпакувати його
unrar x file1.rar – розпакувати rar-архів
unzip file1.zip – розтиснути і розпакувати zip-архів RPM пакети (Fedora, Red Hat і т.і):
zip -r file1.zip file1 file2 dir1 – створити стиснутий zip-архів і помістити в нього файли і/або каталоги
zip file1.zip file1 – створити стиснутий zip-архів

4. Дисковий простір

df -h – відобразити інформацію про змонтовані розділи з відображенням загального, доступного та використовуваного простору
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.)
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 розміром
ls -lSr | more – вивести список файлів і каталогів рекурсивно з сортуванням за зростанням розміру і дозволяє здійснити посторінковий перегляд
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.)

5. Користувачі і групи

chage -E 2005-12-31 user1 — встановити дату закінчення дії облікового запису користувача user1
chattr +S file1 — вказати, що, при збереженні змін, буде виконана синхронізація, як при виконанні команди sync
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'a
chattr +i file1 — зробити файл недоступним для любих змін: редагування, вилучення, переміщення, 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 +u file1 — вказати, що при вилученні файлу його вміст буде збережений и при необхідності користувач зможе його відновити
chgrp group1 file1 — змінити групу-власника файлу file1 на group1
chmod g+s /home/public — встановити SGID-біт каталогу /home/public
chmod g-s /home/public — зняти SGID-біт з каталогу /home/public
chmod go-rwx directory1 — відібрати у групи і всіх інших всі повноваження на каталог directory1.
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
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 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
chown -R user1 directory1 — призначити рекурсивно власником каталогу directory1 користувача user1
chown user1 file1 — призначити власником файлу file1 користувача user1
chown user1:group1 file1 — змінити власника і групу власника файлу file1
find / -perm -u+s — знайти, починаючи від кореня, всі файли з встановленим SUID
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
grpck — перевірити коректність системних файлів облікових записів. Перевіряється файл/etc/group
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
lsattr — показати атрибути файлів
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
passwd user1 — змінити пароль користувача user1 (тільки root)
passwd — змінити пароль
pwck — перевірити коректність системних файлів облікових записів. Перевіряються файли /etc/passwd і /etc/shadow
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

6. Встановлення програмних пакетів

6.1. 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` – очищення 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.

6.2. 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 розв'язування залежностей
yum clean all – очистити rpm-кеш, вилучивши закачані пакети і заголовки

6.3. Встановлення програмних пакетів в Debian, Ubuntu (DEB)

apt – засіб керування пакетами (Debian, Ubuntu і т.п.):
apt help – довідка щодо команди apt
sudo apt update – оновити базу даних репозиторію
sudo apt upgrade – оновити встановлені пакети
s**u**do **apt** full-upgrade – оновлення всіх встановлених пакетів
sudo apt install [package-name] – встановити один пакет
sudo apt install [package-name-1] [package-name-2] ... [package-name-n] – встановити декілька пакетів
sudo apt remove [package-name] – вилучити пакет
sudo apt purge [package-name] – вилучити пакет з конфігураційними файлами
apt search [search-text] – пошук пакета у сховищі
apt show [package-name] – перегляд вмісту пакета
apt list --installed – список встановлених пакетів
apt list --upgradable – список доступних оновлень пакетів
sudo apt autoremove – вилучення невикористовуваних пакетів
apt depends [package-name] – перевірка залежності пакетів
sudo apt reinstall [package-name] – перевстановлення пошкоджених пакетів
apt download [package-name] – завантаження пакету без інсталяції
apt changelog [package-name] – перевірка журналу змін пакету
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

7. 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

8. Файлові системи

8.1. Аналіз файлових систем

badblocks -v /dev/hda1 – перевірити розділ hda1 на наявність bad-блоків
e2fsck -j /dev/hda1 – перевірити/відновити цілісність файлової системи ext3 розділу hda1 з вказівкою, що журнал розміщений там же
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
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

8.2. Створення файлових систем

fdformat -n /dev/fd0 **–** форматування гнучкого диску без перевірки
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
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

8.3. Монтування файлових систем

fuser -km /mnt/hda2 – примусово розмонтувати розділу. Застосовується у випадку, коли розділ зайнятий яким-небудь користувачем
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'
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 "символьні посилання" зробити неможливо. Виконання даної команди зробить копію вмісту /home/user/prg в /var/ftp/user

9. Створення резервних копій (backup)

dd bs=1M if=/dev/hda | gzip | ssh user@ip_addr 'dd of=hda.gz' – зробити «копію зліпок» локального диску в файл на віддаленому комп'ютері через ssh-тунель
dd if=/dev/fd0 of=/dev/hda bs=512 count=1 – відновити MBR з гнучкого диску на /dev/hda
dd if=/dev/hda of=/dev/fd0 bs=512 count=1 – створити копію MBR (Master Boot Record) з /dev/hda на гнучкий диск
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
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
restore -if /tmp/home0.bak – відновити з резервної копії /tmp/home0.bak
rsync -az -e ssh --delete /home/local ip_addr:/home/public – синхронізувати віддалений каталог з локальним каталогом через ssh-тунель із стисканням
rsync -az -e ssh --delete ip_addr:/home/public /home/local – синхронізувати локальний каталог з віддаленого каталогу через ssh-тунель із стисканням
rsync -rogpav --delete /home /tmp – синхронізувати /tmp з /home
rsync -rogpav -e ssh --delete /home ip_address:/tmp – синхронізувати через SSH-тунель
tar -Puf backup.tar /home/user – створити інкрементну резервну копію каталогу '/home/user' у файл backup.tar із збереженням повноважень
tar cf - . | (cd /tmp/backup ; tar xf -) – копіювання одного каталогу в інший із збереженням повноважень і посилань
(**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

10. Робота з CDROM

cd-paranoia -- "-3" – переписати перші три аудіо записи з CD у wav файли
cd-paranoia -B – переписати аудіо записи з CD у wav файли
cdrecord --scanbus – сканувати шину для ідентифікації scsi каналу
cdrecord -v dev=/dev/cdrom cd.iso – записати ISO образ на cdrom
cdrecord -v gracetime=2 dev=/dev/cdrom -eject blank=fast -force – очистити повторно записуваний (rewritable) cdrom
gzip -dc cd_iso.gz | cdrecord dev=/dev/cdrom ——записати стиснутий ISO образ на cdrom
mkisofs -J -allow-leading-dots -R -V "Label CD" -iso-level 4 -o ./cd.iso data_cd – створити iso образ з каталогу
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
mount -o loop cd.iso /mnt/iso – змонтувати ISO образ

11. Використання ресурсів і пристроїв

badblocks -s dev/sda1 – test на недосяжні дискові блоки
cat /proc/cpuinfo – інформація про процесор
cat /proc/devices – інформація про всі пристрої
cat /proc/meminfo – інформація про пам’ять
df – обсяги пам’яті на розділах
dmesg | less – виведення інформації про кільцевий буфер ядра
dmesg – показати log-файл завантаження системи (bootup)
dmidecode – інформація про BIOS
du – кількість дискових блоків в каталогах
free – інформація про використання оперативної пам’яті
free –m – інформація про вільну пам'ять (ОЗП і Swap) у Мбайтах
hdparm -i dev/sda1 – інформація про диски
hdparm -tT dev/sda1 – тест швидкості диску
hdparm /dev/sda – інформація про параметри жорсткого диску
ipcs –m – розподіл сторінок пам’яті
lsblk – інформація про блокові пристрої
lsdev – інформація про встановлені пристрої
lsmod – інформація про завантажені модулі
lsof | less – інформація про поточні відкриті файли і каталоги
lspci - інформація про PCI пристрої
lspci -tv – інформація про PCI пристрої
lsusb -tv – інформація про USB пристрої
modinfo module – розширена інформація про конкретний модуль
slabtop – використання кеш пам’яті ядром
top – динамічна інформація про використання оперативної пам’яті
vmstat -d – статистика про введення-виведення жорсткого диску
vmstat 1 – інформація про використання віртуальної пам’яті за заданий період часу

12. Мережа (LAN і WiFi)

arp –v – інформація з кешу ARP
bmon – монітор пропускної здатності то оцінки швидкості мережі
dhclient eth0 – активувати інтерфейс eth0 в dhcp-режимі
dig – DNS інформацію
echo "1" > /proc/sys/net/ipv4/ip_forward – дозволити пересилання пакетів (forwarding)
ethtool eth0 – інформація про ethernet карти
ethtool eth0 – відобразити статистику інтерфейсу eth0 з виведенням такої інформації, як підтримувані і поточні режими з'єднання
export http_proxy=http://your.proxy:port – змінити значення змінної оточення http_proxy, для використання Інтернету через proxy-сервер
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
hostname – відобразити мережене ім'я локальної машини
ifconfig eth0 -promisc – відключити promiscuous-режим на інтерфейсі 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 – показати конфігурацію мережевого інтерфейсу eth0 (MAC адресу і IP-адресу TCP/IP з'єднання)
ifstat – статистика мережевих інтерфейсів
ifdown eth0 – деактивувати (опустити) інтерфейс eth0
ifup eth0 – активувати (підняти) інтерфейс eth0
ip addr show – отримання інформації про мережеві з'єднання
ip link show – відобразити стан усіх інтерфейсів
ip route show – виведення інформація про маршрутизацію
ip route – тестування шлюзу з таблиці маршрутизації
iwconfig eth1 – показати конфігурацію бездротового мережевого інтерфейсу eth1
iwlist scan – просканувати ефір на предмет доступності бездротових точок доступу
lsof –i – список усіх відкритих портів у Інтернет
mii-tool eth0 – відобразити статус і тип з'єднання для інтерфейсу eth0
netstat -an | grep LISTEN – список усіх відкритих портів
netstat -rn – вивести локальну таблицю маршрутизації
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
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 –i – статистика про мережеві інтерфейси
netstat –s | less – статистика про пересилання мережевих пакетів
netstat -anp --udp --tcp | grep LISTEN – список застосувань, які відкривають порти
netstat –tup – активні з'єднання з Інтернетом
nmap 192.168.2.100 – сканування портів
ping 192.168.2.1 – перевірка доступності IP-адреси
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
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)
route -n – вивести локальну таблицю маршрутизації
service network status – інформація про мережу
socklist – виведення всіх відкритих сокетів
tcpdump tcp port 80 – відобразити увесь трафік на TCP-порт 80 (звичайно – HTTP)
wall Привіт – посилання повідомлення "Привіт" на термінали інших користувачів

13. Microsoft Windows networks(SAMBA)

mount -t smbfs -o username=user,password=pass //winclient/share /mnt/share – змонтувати smb-ресурс, наявний на windows-машині, в локальну файлову систему
nmblookup -A ip_addr – дозволити netbios-ім'я nbtscan ставити за замовчуванням не у всіх системах. Можливо, прийдеться додатково встановлювати вручну. nmblookup включений у пакет samba.
nbtscan ip_addr – сканування IP-адрес
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

14. 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 підключення 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»
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
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.

15. 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
bg – продовжити виконання фонового процесу, якщо він зупинений натисканням клавіш <Ctrl+Z>
fg – вивести процес з фонового режиму

free -m – показати стан оперативної пам'яті в мегабайтах
fuser – пошук процесів, які мають відкриті файли або сокети
hash – інформація про hash таблиці
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).
jobs – інформація про задачі
kill -1 98989 або kill -HUP 98989 – заставити процес з PID 98989 перечитати файл конфігурації
kill -9 98989 або kill -KILL 98989 – «убити» процес з PID 98989 «на смерть» (без дотримання цілісності даних)
kill -TERM 98989 – коректно завершити процес з PID 98989
killall – перервати виконання процесу за іменем процесу
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
lsof -p 98989 – відобразити список файлів, відкритих процесом з PID 98989
lsof /home/user1 – відобразити список відкритих файлів з каталогу /home/user1
nice – задати пріоритет процесу перед його запуском
pgrep – інформація про ID процесів
ps -e -o pid,args --forest – вивести PID'и і процеси як дерево
ps -eafw – відобразити запущені процеси, використовувані ними ресурси та іншу корисну інформацію (одноразово)
ps -x & – запуск фонового процесу . При завантаженні системи, специфічні процеси ("демонами"), завантажують у фоновий режим. Їх розміщують у каталозі /etc/rc.d/init.d/.
ps, (**ps** -aux) – виводить інформацію про виконувані процеси користувачів
pstree – відобразити дерево процесів
renice – змінити пріоритет виконуваного процесу
renice – змінити пріоритет фонової задачі, якою володіє користувач
smartctl -A /dev/hda – контроль стану жорстокого диску /dev/hda через SMART
smartctl -i /dev/hda – перевірити доступність SMART на жорсткому диску /dev/hda
strace -c ls >/dev/null – вивести список системних викликів, створених і отриманих процесом ls
strace -f -e open ls >/dev/null – вивести виклики бібліотек
tail /var/log/dmesg – вивести десять останніх записів з журналу завантаження ядра
tail /var/log/messages – вивести десять останніх записів з системного журналу
top – динамічно відобразити інформацію про запущені процеси, використовувані ними ресурси та іншу корисну інформацію (з автоматичним обновленням даних)
watch -n1 'cat /proc/interrupts' – відображати переривання в режимі реального часу
Комбінації клавіш для керування завданнями:
Ctrl+Z – призупинити виконання завдань
Ctrl+C – завершити виконання завдань

16. Каталоги файлової системи Linux

Кореневий каталог ″/″ ОС Linux містить підкаталоги із стандартними іменами, які визначаються стандартом (Filesystem Hierarchy Standard).

Підкаталоги кореневого каталогу

Каталог	Опис
/	Кореневий каталог з якого починається файлова система.
/bin, /usr/ 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 ядро (vmlinuz) і файли, необхідні для найпершого етапу - завантаження ядра (boot loader). 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, /drv/fd0 - пристрій першого гнучкого диску, /dev/ttyN - віртуальна консолі, 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. /dev/sda - пристрій першого жорсткого диску. Тут подані всі пристрої, які розуміє ядро.
/etc	Каталог для системних конфігураційних файлів. Тут зберігається інформація про специфічні налаштування даної системи: інформація про зареєстрованих користувачів, доступні ресурси, налаштування різних програм. /etc/passwd - файл у якому зберігаються облікові записи користувачів. /etc/fstab - файл містить таблицю пристроїв, які монтуються при завантаженні системи. Файл визначає драйвери системних дисків. /etc/hosts - у файлі перераховані імена хостів мережі та IP-адреси, які внутрішньо відомі системі. /etc/init.d - каталог містить сценарії, які запускають різні системні служби під час завантаження.
/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. Загалом, це єдине місце, де користувачам дозволено записувати файли.
/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.ін. 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 спільно використовувані 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.). Каталог /media використовується механізмами автоматичного монтування пристроїв у сучасних дистрибутивах Linux, орієнтованих на робочий стіл. Побачити, які пристрої та точки монтування використовуються, можна командою mount.
/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. У цьому каталозі є група пронумерованих записів, які відповідають усім процесам, що виконуються в системі. Крім того, існує ряд іменованих записів, які дозволяють отримати доступ до поточної конфігурації системи. Багато з цих записів можна переглянути, наприклад /proc/cpuinfo.
/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, /usr/ bin	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. /usr/bin - програми для користувачів системи. /usr/share - файли підтримки для системи X Windows. /usr/share/dict - словники для перевірки орфографії. /usr/share/doc - різні файли документації в різних форматах. /usr/share/man - сторінки довідкової системи man Прикладних програм у сучасних системах звичайно встановлено дуже багато, тому цей розділ файлової системи може бути дуже великим.
/usr/ local	/usr/local і його підкаталоги використовуються для встановлення програмного забезпечення та інших файлів для використання на локальній машині. Насправді це означає, що програмне забезпечення, яке не є частиною офіційного дистрибутиву (яке зазвичай міститься в /usr/bin), потрапляє сюди. Для встановлення програм у системі, їх слід встановити в один із каталогів /usr/local. Найчастіше вибирається каталог /usr/local/bin.
/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. /var/log - містить файли журналу. Вони оновлюються під час роботи системи. Ці файли можна час від часу переглядати файли, щоб контролювати стан системи. /var/spool - каталог використовується для зберігання файлів, які знаходяться в черзі для певного процесу, наприклад поштових повідомлень і завдань друку. Коли пошта користувача вперше надходить до локальної системи (якщо вона має локальну пошту, що рідко буває на сучасних машинах, які не є поштовими серверами), повідомлення спочатку зберігаються в /var/spool/mail

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.

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.

5. ОБОЛОНКА BASH. РОЗШИРЕННЯ ВИРАЗІВ

Мета. Вивчення Bash оболонки Bsdh, розширення виразів, регулярних виразів

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.

План.

1. Оболонка 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 використовують ключові слова оболонки для керування виконанням команд ОС. Сценарії виконуються оболонкою порядково і тому швидкість виконання їх невисока.

1.1. Послідовність оброблення сценаріїв оболонкою Bash

Терміни.

Метасимвол. Символ, який, якщо не взятий в лапки, розділяє слова. До метасимволів відносяться: ***space*** *(пропуск)*, ***tab*** *(табуляція)*, ***newline*** *(новий рядок)* або один із таких символів: `'|'`, `'&'`, `';'`, `'('`, `')'`, `'<'`, `'>'`.

Токен. Послідовність символів, яку оболонка Bash вважає одним цілим. Це може бути *слово* або *оператор*.

Слово. Послідовність символів, яка розглядається оболонкою Bash як єдине ціле. Слова не можуть містити метасимволи без лапок.

Оператор. Оператор керування або оператор перенаправлення. Оператори містять принаймні один метасимвол без лапок.

Оператори керування: `newline, '||', '&&', '&', ';', ';;', ';&', ';;&', '|', '|&', '(', ')'.`

Оператор перенаправлення:<, >, <>, <<, <<-, >>, <<<, `<&, >&`.

Зарезервоване слово. Слово, яке має особливе значення для оболонки Bash. Зарезервовані слова позначають початок та завершення складених команд оболонки.

`if`	`then`	`elif`	`else`	`fi`	`time`
`for`	`in`	`until`	`while`	`do`	`done`
`case`	`esac`	`coproc`	`select`	`function`	
`{`	`}`	`[[`	`]]`	`!`	

Послідовність оброблення сценарію оболонкою:

1. Читання вхідних даних з файлу, з терміналу користувача або символьного рядка аргументу (параметр виклику -c).

2. Розбиття вхідних даних на токени (слова та оператори), з врахуванням лапок. Ці токени розділені метасимволами. Розгортання псевдонімів (alias) виконується на цьому кроці.

3. Розбір токенів на прості та складені команди.

4. Виконання різних розширень оболонки, розбиття розширених токенів на списки імен файлів, команд і аргументів.

5. Виконання необхідних перенаправлень і вилучення операторів перенаправлень та їхніх операндів зі списку аргументів.

6. Виконання команди.

7. Очікування завершення команди та збереження її статус виходу.

1.2. Використання лапок

Лапки використовується для вилучення особливого значення певних символів або слів в оболонці. Лапки можна використовувати, щоб вимкнути спеціальну обробку спеціальних символів, запобігти розпізнаванню зарезервованих слів як таких і запобігти розширенню параметрів.

Є три механізми використання лапок: символ екранування, одинарні лапки та подвійні лапки.

Символ екранування – це зворотна похила риска \ без лапок. Вона зберігає буквальне значення наступного наступного символу, за винятком нового рядка (`\n`).

Якщо взяти символи в одинарні лапки (`'...'`), зберігається буквальне значення кожного символу в лапках. Одинарні лапки не можуть бути між одинарними лапками, навіть якщо перед ними стоїть зворотна похила риска.

Узяття символів у подвійні лапки (`"..."`) зберігає літеральне значення всіх символів у лапках, за винятком `'$', '`', '\'`. Зворотна похила риска зберігає своє спеціальне значення лише тоді, коли за ним стоїть один із таких символів: `'$', '`', '"', '\'` або новий рядок. У подвійних лапках зворотні похилі риски, за якими йде один із цих символів, вилучаються.

Символи "$" і "`" зберігають своє особливе значення в подвійних лапках. Спеціальні символи `*` та `@` у подвійних лапках також мають особливе значення.

1.3. Змінні Bash і користувача

У Bash є два типи змінних:

- змінні визначені користувачем;
- змінні оболонки Bash.

Змінні оболонки Bash встановлюються системою і мають спеціальне значення. Змінні середовища записуються у файлі /etc/profile або ~/.bash_profile. Змінні оболонки створюються при першому запуску оболонки Bash і забезпечують інформацію про поточну сесію, а також можуть використовуватися для контролю деяких параметрів оболонки.

Змінні оболонки Бйорна: CDPATH, HOME, IFS, MAIL, MAILPATH, OPTARG, PS1, PS2.

Змінні оболонки Bash: _, BASH, BASHOPTS, BASHPID, BASH_ALIASES, BASH_ARGC, BASH_ARGV, BASH_COMMAND, BASH_COMPAT, BASH_ENV, BASH_EXECUTION_STRING, BASH_LINENO, BASH_LOADABLES_PATH, BASH_REMATCH, BASH_SOURCE, BASH_SUBSHELL, BASH_VERSINFO, BASH_VERSION, BASH_XTRACEFD, CHILD_MAX, COLUMNS, COMP_CWORD, COMP_LINE, COMP_POINT, COMP_TYPE, COMP_KEY, COMP_WORDBREAKS, COMP_WORDS, COMPREPLY, COPROC, DIRSTACK, EMACS, ENV, EPOCHREALTIME, EPOCHSECONDS , EUID, EXECIGNORE, FCEDIT, FIGNORE, FUNCNAME, FUNCNEST, GLOBIGNORE, GROUPS, HISTCMD, HISTCONTROL, HISTFILE, HISTFILESIZE, HISTIGNORE, HISTSIZE, HISTTIMEFORMAT, HOSTFILE, HOSTNAME, HOSTTYPE, IGNOREEOF, INPUTRC, INSIDE_EMACS, LANG, LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_NUMERIC, LC_TIME, LINENO, LINES, MACHTYPE, MAILCHECK , MAPFILE, OLDPWD, OPTERR, OSTYPE, PIPESTATUS, POSIXLY_CORRECT, PPID , PROMPT_COMMAND, PROMPT_DIRTRIM, PS0, PS3, PS4, PWD, RANDOM, READLINE_LINE, READLINE_MARK, READLINE_POINT, REPLY, SECONDS, SHELL, SHELLOPTS, SHLVL, SRANDOM, TIMEFORMAT.

Команди declare, set, typeset, printenv без параметрів виводять значення всіх змінних оболонки. Приклад значень деяких з них:

```
$ set
```

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 поточного комп’ютера
HOSTTYPE – архітектура поточного комп’ютера
HOME – абсолютний шлях до домашнього каталогу користувача
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 – рядок основного повідомлення командного рядка
PS2 - рядок вторинного повідомлення командного рядка
PS3 - рядок третинного повідомлення командного рядка
PS4 – повідомлення на початку кожного рядка для команди trace або запуску сценарію з ключем -x
PWD – поточний робочий каталог
RANDOM – генератор випадкового цілого число в діапазоні 0-32767

SECONDS – час роботи сценарію в секундах
SHELL – вибрана оболонка
TERM – тип емуляції терміналу
UID – ідентифікатор користувача в /etc/password
USER - список каталогів у яких оболонка Bash буде шукати виконувані файли

Змінні визначені користувачем створюються при призначенні їм значень.

Усі змінні Bash записані 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 є контейнерами, які можуть містити наступні значення:
- числові;
- символи;
- символьні стрічки;
- змінні константи;
- змінні масиви.

Присвоїти значення змінній можна за допомогою інструкції ”=” (без пропусків перед і після =) або команди `let`. Приклад присвоєння змінним цілочислового, дійсного і стрічкового значень:

```
$ a=1; b=2.0; c=3e-4; d=”Привіт Bash”
$ echo $a $b $c $d
1 2.0 3e-4 Привіт Bash
```

Команда `let` дозволяє присвоїти значення одній або декільком змінним:

```
$ let a=1
$ echo $a
1
```

```
$ let "a=1" "b=2" "c=3"
echo $a $b $c
1 2 3
```

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
$ var
$ echo :$var:
::
```

Відміна значення змінної

```
$ a=
$ unset b
```

Команда `set` вбудована команда оболонки, яка відображає і встановлює імена і значення змінних оболонки і середовища Linux:

```
$ set <параметри> <аргументи>
```

Запуск команди без параметрів і аргументів виводить список усіх налаштувань

```
 $ set | head -15
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extqu
ote:force_fignore:globasciiranges:histappend:interactive_comments:login_shell:prog
comp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=([0]="0")
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_VERSINFO=([0]="2" [1]="10")
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="5" [1]="0" [2]="17" [3]="1" [4]="release" [5]="x86_64-pc-
linux-gnu")
BASH_VERSION='5.0.17(1)-release'
COLUMNS=120
DIRSTACK=()
DISPLAY=:0
EUID=1000
```

Отримання значень складових символьної стрічки командою `set`. При цьому будуть втрачені оригінальні позиційні параметри сценарію, з якими він був викликаний.

```
$ x="Привіт Bash 2025 !"
$ set -- $x
$ a=$1; b=$2; c=$2 d=$3
$ echo "$a"
$ echo "&b = $c"
$ echo "$d"
Привіт
Bash = 2025
!
```

Команда declare (синонім **typeset)**.

Змінні Bash мають **атрибути**, які можуть бути призначені префіксом (-) або відмінені префіксом (+) за допомогою команди `declare`.

Команда `declare (typeset)` призначає змінним атрибути:

```
declare [опції] [ім'я-змінної] "[значення]"
```

Опції для функцій:

-f оголошення функції

-F виводить імена функції і атрибути

-g застосувати глобальну область видимості для всіх змінних всередині функції

Опції для змінних:

-a змінна є індексований масив. Її атрибути змінити не можна

-A змінна є індексований масив. Її атрибути змінити не можна

-i змінна є цілим число. Відміна атрибуту +i

-l ім'я змінної складається тільки з малих букв. Відміна атрибуту +l

-n змінна стає посиланням на іншу змінну. Відміна атрибуту +n

-r змінна тільки для читання. Відміна атрибуту +r

-t якщо використовується з функціями, елемент успадковує перехоплення DEBUG і RETURN від батьківської оболонки. Відміна атрибуту +t

-u ім'я змінної складається тільки з великих букв. Відміна атрибуту +u
-x експорт змінною до дочірнього процесу. Відміна атрибуту +x
-p перевірка успішності створення змінної

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).

```
$ declare –l var=”ABC”; echo $var
abc
```

Символи верхнього регістру: `declare -u variable` (всі символи змінної конвертуються у верхній регістр)

```
$ declare –u var=”aBc”; echo $var
ABC
```

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”
Старе місто
$ COMPANY=”Нове місто”
Sh: Змінна призначена лише для читання
```

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` (експортована змінна успадковується любою підоболонкою або дочірнім процесом).

```
$ declare -x COST=”/home/cvs/cvsroot”
```

Створені змінні існують до завершення сценарію або знищення їх вбудованою командою `unset`

```
$ unset var
```

1.4. Запуск оболонки Bash і виконання сценаріїв

Поточна оболонка:

```
$ echo $SHELL
/bin/bash

$ echo ”$0”
/bin/bash
```

Пошук шляху до оболонки:

```
$ type -a bash
bash is /usr/bin/bash
bash is /bin/bash
```

```
$ type -a csh
bash is /usr/bin/csh
bash is /bin/csh
```

Зміна оболонки

```
$ chsh -s /bin/csh
```

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 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 (vim, nano. kwrite) і зберігається у файлі з розширенням .bash або `.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 u+x hello.sh
$ ./hello.sh
Привіт Bash!
```

Виконати сценарій у *режимі налагодження* можна запуском Bash з параметром `-x`

```
$ bash -x hello.sh
+ for i in 1 2
+ echo 1
1
+ for i in 1 2
+ echo 2
2
```

1.5. Структура Bash-сценарію

При написанні Bash сценарію рекомендується використовувати наступну структуру:

- 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 –os xtrace  # вивести на консоль кожну команду перед її виконанням
```

В заголовку можуть бути як однорядкові так і багаторядкові коментарі

```
#!/bin/bash
# Однорядковий коментар
: '
Багаторядковий
коментар
'
echo "   Привіт \
         Bash \
         програмістам"
$ bash hello.bash
Привіт          Bash          програмістам
```

Глобальні оголошення – виконують присвоєння змінним або оголошують змінні з атрибутами, які доступні у всьому сценарії і його функціях:

```
# Global Declarations
s="Привіт світ";i=10;f=3.14;
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
```

Перевірка наявності необхідних команд і файлів 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 реалізує функціонал всього сценарію, наприклад

```
# Примусовий запис змінених блоків на диск при відсутності користувачів
USERS=`who | wc -l`
if [ $USERS -eq 0 ] ; then
sync
fi
```

Завершення сценарію - очищаються, при наявності, тимчасові файли і задається код завершення сценарію:

```
exit 0   # нормальне завершення
```

1.6. Команди Linux і 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 реалізована в 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 реалізації 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 реалізації в оболонці Bash кожної команди.

В одному рядка можна вводити *одну* або *декілька* команд (розділивши їх «;»):

```
$ date
$ pwd
$ date; pwd   # після останньої команди не ставиться «;».
```

Можна передати результат виконання одної команди на вхід іншої команди з використанням символу перенаправлення (конвеєра):

```
command1 | command2
```

Можна організувати список команд розділених одним із символів `';', '&', '&&', '||'`, який завершується символами `';', '&'`, або а `newline`.

```
command1 && command2
```

Команда `command2` виконується при умові завершення команди `command1` з кодом 0 (успіх).

```
command1 || command2
```

Команда `command2` виконується при умові завершення команди `command1` з кодом не 0 (не успіх).

2. Групування команд

Bash надає два способи групування команд у список, які потрібно виконати як одиницю. Коли команди згруповано, переспрямування можна застосувати до всього списку команд. Наприклад, вихід усіх команд у списку може бути перенаправлений до одного потоку.

```
( список )
```

Розміщення списку команд у круглих дужках змушує оболонку створити підоболонку і кожна з команд у списку виконується в цьому середовищі. Оскільки *список* виконується у підоболонці, значення змінних втрачаються після завершення підоболонки. Кожна підоболонка має свої змінні середовища.

```
$ ( sleep 5 ; printf "%s\n" "Slept for 5 seconds" )  # підоболонка
Slept for 5 seconds

$ printf "%d\n" "$COUNT"
10
$ ( COUNT=20 ; printf "%d\n" "$COUNT" ) # підоболонка
20

$ printf "%d\n" "$COUNT"
10
```

Підоболонки часто використовуються з символом перенаправлення |. У цьому випадку на вхід підоболонки подається результат виконання попередньої команди:

```
# 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
```

Розміщення списку команд між *фігурними дужками* призводить до того, що список буде виконано в поточному контексті оболонки. Підоболонка не створюється. Крапка з комою (або новий рядок) у наступному *списку* команд є обов’язковою.

```
{ список; }
```

```
$ declare -ix COUNT=15
$ { COUNT=10 ; printf ”%d\n” ”$COUNT” ; }  # група
10
```

Результат роботи однієї команди можна передати в іншу використовуючи символ конвеєра ”|”. Наприклад, результат виконання команди `ls` стає входом для команд у фігурних дужках:

```
$ ls -l | { while read FILE do ; echo “$FILE” done }
```

Між цими двома конструкціями є різниця. *Круглі дужки є операторами*, і розпізнаються оболонкою як окремі токени, навіть якщо вони *не відокремлені від списку* команд символами пропуску. *Фігурні дужки є зарезервованими словами* і тому *повинні відділятися від списку команд символами пропуску* або іншими метасимволами.

Статус завершення обох конструкцій є статусом завершення списку.

3. Розширення (розгортання) виразів

Розширення (розгортання) виконується в командному рядку після того, як його було розділено на токени. Є сім видів розширень, які виконуються у наступному порядку:

- розширення дужки;
- розширення тильди;
- розширення параметра і змінної;
- підставлення (заміна) команди;
- арифметичне розширення;
- розбиття слова;
- розширення імені файлу.

Після виконання всіх розширень лапки вилучаються.

3.1. Розширення дужок

Розширення (розгортання) дужок – це механізм, за допомогою якого можна генерувати довільні рядки. Синтаксис:

```
префікс{a,b,c}суфікс
```

Префікс додається до кожного рядка, що міститься в дужках, а суфікс додається до кожного результуючого рядка, розгортаючись зліва направо.

```
$ echo a{a,b,c}x
aax abx acx
$ echo /home/user/user{1,2,3{1,2}}
/home/user/user1 /home/user/user2 /home/user/user31 /home/user/user32
$ echo {1..5}
1 2 3 4 5
$ echo {a..d}
a b c d
$ mkdir a{1..3}
$ mkdir a(1..3/b(1..2)
$ cd ~/a1/b1
$ pwd
homeuser/a1/b1
```

3.2. Розширення тильди ~

Домашній каталог користувача:

```
$ echo ~
home/user
```

Поточний каталог:

```
$ echo ~+   (або echo $PWD)
home/user/doc2
```

Попередній поточний каталог:

```
$ echo ~-  (або echo $OLDPWD)
home/user/doc1
```

Переміщення у каталог користувача:

```
$ cd ~
home/user/
```

Переміщення між двома каталогами:

```
$ cd -
home/user/
$ cd -
home/user/Downloads
```

3.3. Розширення параметра і змінної

Символ "$" вводить розширення параметрів, підставлення команд або арифметичне розширення. Ім'я параметра або символ, що розгортається, може бути укладений у фігурні дужки, які є необов'язковими, але служать для захисту змінної, що розгортається, від символів, які слідують безпосередньо за нею і які можуть бути інтерпретовані як частина назви.

Основною формою розширення параметра є ${*параметр*}. Підставляється значення *параметра* або посилання на масив. Фігурні дужки є обов'язковими, якщо *параметр* є позиційним параметром із більш ніж однією цифрою або коли за *параметром* стоїть символ, який не слід інтерпретувати як частину його імені.

Якщо перший символ *параметра* є знаком оклику (!), то це непряме посилання:

```
$ name="var"
$ var=5
```

```
$ echo ${!name}
5
```

Винятком є розширення ${! *префікс* *} і ${! *ім'я* [@]}, описане нижче.

Якщо *параметр* не встановлено або має нульове значення, замінюється розширення *слова.* В іншому випадку значення *параметра* замінюється.

```
$ {parameter:-word}
```

```
$ a=123
$ echo ${a-unset}
123
```

Якщо *параметр* не встановлено або має нульове значення, розширення *слова* призначається *параметру* . Потім замінюється значення *параметра* . Позиційні параметри та спеціальні параметри не можуть бути призначені таким чином.

```
$ {parameter:=word}
```

```
$ a=
$ : ${a:=DEFAULT}
$ echo $a
DAFAULT
```

Розширення підрядка починається із символу визначеного `offset` і довжиною `length`.

```
$ {parameter:offset}
$ {parameter:offset:length}
```

```
$ string=01234567890abcdefgh
$ echo ${string:7}
7890abcdefgh

$ echo ${string:7:-2}
7890abcdef
```

```
$ set -- 01234567890abcdefgh
$ echo ${1:7:2}
78

$ arr=(1 2 3 4)
$ echo ${arr[@]:2}
3 4
```

Наступні приклади ілюструють розширення підрядка за допомогою позиційних параметрів:

```
# set -- задає позиційні параметри
$ set -- 1 2 3 4 5 6 7 8 9 0 abcdefgh
$ echo ${@:7}
7 8 9 0 abcdefgh

$ echo ${@:7:2}
7 8
```

```
$ echo ${@: -7:2}
bc
```

```
$ echo ${@:0}
./bash 1 2 3 4 5 6 7 8 9 0 abcdefgh
```

```
$ echo ${@:0:2}
```

```
./bash 1
```

3.3.1. Підставлення і розширення змінної

Ім’я змінної задає місце для зберігання значення. Для отримання значення змінної `var` використовують посилання на її ім’я

```
$var
${var}
```

Посилання `$var` є спрощеним синтаксисом від загального розширення змінної `${var}`, яке дозволяє виконувати перетворення над рядком змінної.

```
${var^} перетворення першого символу на велику літеру
${var^^} перетворення всіх символів на велику літеру
${var,} перетворення перший символ на малу літеру
${var,,} перетворення всіх символів на малу літеру
${#var} кількість символів у змінній
${var:offset:length} виділення length символів з позиції offset
```

```
$ var=23
$ echo var
var
$ echo $var
23
$ echo ${var}
23
$ printf ”%s\n” ”Вартість ${var} грн”
Вартість 23 грн
```

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 з використанням символу "**!**":

```
$ a=b
$ b=”hello”
$ echo {!a}
hello
```

3.4. Підставлення виводу команди

Оболонка підставляє замість команди стандартний вивід команди за допомогою круглих дужок (). Синтаксис:

```
$(Linux_command)
```

```
$ echo $(date)
Sun 25 Sep 2022 05:54:47 PM EEST
```

```
$ echo $(ls)
1.sh 2.sh 3.sh
```

```
$ echo ”$(ls)”
1.sh
2.sh
3.sh

$ declare NUMBER_OF_FILES
$ NUMBER_OF_FILES=$(ls -l | wc –l)
$ printf ”%d” ”$NUMBER_OF_FILES”
14
```

3.4.1. Підставлення процесу

Підставлення процесу використовує файл `dev/fd<n>` для надсилання результатів процесу на вхід `stdin` іншого процесу. Синтаксис:

```
<(список)
>(список)
```

Процес виконується асинхронно і його вхід або вихід з’являється як ім’я файлу. Це ім’я файлу передається поточній команді як результат підставлення. Між символами <, > і дужкою (не має бути символу пропуску. Якщо буде символ пропуску то символи <, > інтерпретуються як перенаправлення. Підставлення процесу підтримується в системах де є іменовані канали FIFO або /`dev/fd` метод іменування відкритих каналів.

```
$ cat <(true)
/dev/fd/63

$ echo <(true) <(true)
/dev/fd/63 /dev/fd/62

$ cat <(ls -l)
```

аналог до

```
$ ls -l | cat

; wc <(cat myfile)

$ cat myfile | wc
```

Порівняння вмісту двох каталогів

```
diff <(ls $first_directory) <(ls $second_directory)
```

3.6. Розбиття слів

Оболонка сканує результати розширення параметрів, підставлення команд і арифметичного розширення, які не відбувалися у подвійних лапках для розбиття слів.

Оболонка розглядає кожен символ $IFS, як роздільник і розбиває результати інших розширень на слова, використовуючи ці символи. Якщо IFS не встановлено або його значення дорівнює <space>|<tab>|<newline> за замовчуванням, тоді послідовності <space>, <tab>, і <newline>на початку та в кінці результатів попередніх розгортань ігноруються, а будь-яка послідовність IFS символів не на початку чи в кінці служить для розбиття слів.

4. 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 (ERE).

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 зворотною 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.

Символи регулярних виразів:

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
```

5. Розширення імен файлів

У Bash символи підставлення в імена файлів називають розширенням шляху (англ. globbing). При розширенні імені шляху використовують спеціальні символи `*`, `?` і `[...]`, які вводяться як частину шляху в командах, наприклад:

Після поділу слів, якщо тільки не було встановлено параметр `-f`, Bash сканує кожне слово на наявність символів `*`, `?` і `[`. Якщо один із цих символів з'являється без лапок, то слово розглядається як *шаблон* і замінюється впорядкованим за алфавітом списком назв файлів, які відповідають шаблону.

Будь-який символ, який з'являється у шаблоні, окрім спеціальних символів шаблону, описаних нижче, збігається сам із собою. Спеціальні символи шаблону повинні бути взяті в лапки, якщо вони повинні бути зіставлені буквально.

Спеціальні символи шаблону мають такі значення:

* як єдиний шаблон, відповідатиме всім файлам, каталогам і підкаталогам поточного каталогу (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
```

/* як єдиний шаблон, відповідатиме всім каталогам і підкаталогам поточного каталогу;

** як єдиний шаблон, відповідатиме всім файлам, каталогам і підкаталогам файлової системи;

/** як єдиний шаблон відповідатиме всі каталогам і підкаталогам файлової системи.

? любий одиничний символ.

```
$ echo t?.sh
t2.sh
```

[…] співпадіння із любим із символів, заданих у шаблоні: [abcxyz], [a-cx-z0-1].

```
$ ls -l [a-c]*
```

^, ! – співпадіння з символами, крім заданих.

```
$ ls -l [^a-c]
$ ls -l [!a-c]
```

[:class:] де класи - alnum, alpha, ascii, blank, cntrl, digit, graph, lower, print, punct, space, upper, word, xdigit.

```
ls file*      # усі файли які імена яких починаються на file
ls file[0-9]  # файли file0, file1,… file9
ls file*[0-9] # усі файли які імена яких починаються на file і закінчуються
                цифрами 0,...,9
ls file[0-9]* # усі файли які імена яких починаються на file0, file1,...
ls ???        # усі файли які імена яких мають три символи
ls [!B]*      # усі файли, імена яких не починаються із символу B
```

Крім звичайних символів підставлення, які підтримує bash є і інші форми розширення імен файлів (англ. extended globbing), який стає доступний за опцією

```
$ shopt -s extglob
```

При використанні команди **shopt -s extglob** Bash виконує додаткові розширення імен файлів:

- ?(*pattern-list*)— збіг з нуль або одним входженням шаблоном
- *(*pattern-list*)— збіг з нуль або більше входжень шаблонів
- +(*pattern-list*)— збіг з одним або декількома входженнями шаблонів
- @(*pattern-list*)— збіг з одним входженням шаблону
- !(*pattern-list*)— Збіг з будь-чим, окрім одного з заданих шаблонів

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
```

Тонкий момент розширення імені файлу, полягає в тому, що це робить Bash, а не операційна система чи програма, яка виконується. Програма ніколи не бачить символи розширення, тільки bash замінює розширення в командний рядок перед запуском програми. Якщо програма виконується не через bash, а передається в `exec()`, то жодні символи розширення в командному рядку не розгортатимуться.

Додаток.
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
$# – число аргументів переданих у сценарій
$? – код завершення останньої виконаної команди
$$ – ідентифікатор процесу (PID) самого сценарію
$! – ідентифікатор процесу (PID) останнього завдання запущеного на задньому плані
$0 – ім'я файлу поточного сценарію
(...) – група команд
{...} – група команд
{ } – 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 коментарію.

Все що слідує за цим символом, є коментарієм, 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 коментарію'
echo Тут екранований символ \# також не є початком коментарію
echo А тут # означає коментар
echo ${PATH#*:} # Підставлення параметрів, символ # не є початком коментарію.
echo $(( 2#101011 )) # Основа системи числення, символ # не є початком
коментарію
```

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 файлу. Якщо ім'я файлу 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 одиничний 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 арифметичних виразів. Не дивлячись 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
```

\ – зворотна 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, ОС Linux. При використанні в арифметичних операціях, означає ділення.

` – підставлення команд (лівонахилений апостроф) (не рекомендується).

Можуть використовуватися для присвоєння змінній, результатів виконання системних команд. 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 неіснуючого 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 степеня.

? – перевірка умови.
В деяких випадках використовується для перевірки виконання умови.
При використанні в конструкціях з подвійними дужками задає тернарний оператор мови С.

```
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 розширення змінних або 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

```
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

```
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”
```

виведе

```
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
} < $file # читання рядків з файлу fstab
echo ”$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 цілочисловий 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”
echo $((a++)) # після-інкремент значення змінної $a, в стилі мови C
echo $((a--)) # після-декремент значення змінної $a, в стилі мови C
echo $((++a)) # перед-інкремент значення змінної $a, в стилі мови C
echo $((--a)) # перед-декремент значення змінної $a, в стилі мови C
```

також можна використати тернарні оператори, згадані раніше :

```
echo $((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” ]]; then echo $a; fi
if [ ”$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 ho
```

| – 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 потрібно забувати про значення змінних, наприклад:

```
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 поточний 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.

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. Змінні оболонки Bash і змінні користувача. Оголошення і використання змінних.
6. Розширення дужок
7. Розширення параметру і змінної
8. Розширення команд і змінної
9. Розбиття слів
10. Розширення імен файлів
11. Регулярні вирази у Bash.

6. ТИПИ ДАНИХ І ОПЕРАТОРИ

Мета. Вивчення типів даних і операторів Bash.

Вступ. Оператори Bash реалізують умови галужень, оцінювання умов порівняння, арифметичні вирази, цикли, маніпуляції із стрічками символів.

План.

1. Типи даних
2. Оператори Bash
   2.1. Арифметичні оператори
   2.2. Оператори присвоєння
   2.3. Оператори відношень
   2.4. Логічні оператори
   2.5. Порозрядні оператори
   2.6. Тернарний оператор
3. Арифметичні вирази
   3.1. Команда let
   3.2. Арифметичні вирази у круглих дужках ((...))
   3.3. Одинарні круглі дужки (...)
   3.4. Команда expr для виконання арифметичних виразів
   3.5. Команда bc для арифметичних виразів з плаваючою крапкою
4. Вирази умов
   4.1. Вирази з операторами if, if-then, if-then-else
   4.2. Оцінювання умов порівняння командою test, [...]
   4.3. Подвійні квадратні дужки
5. Оператор case
6. Оператори циклів
7. Маніпуляції стрічками

1. Типи даних

Кожна мова програмування має систему типів. З усіх мов сценаріїв система типів Bash є, найбільш незрозумілою та схильною до помилок. Це настільки непомітно, що користувачі можуть бути введені в оману, вважаючи, що Bash є «нетиповим». Але, як і будь-яка мова, Bash має свої типи.

Система типів дуже проста. Їх всього три: символьна стрічка (string), цілі числа і, головне, списки. Однак складність виникає через два фактори:Система типів дуже проста. Їх всього три: рядок, цілі числа і, головне, списки . Однак складність виникає через два фактори:

• Скрите типування. Тип значення визначається під час виконання на основі його контексту. Таким чином, 1 може бути або ціле число 1, або стрічка «1».

• Оцінювання. Bash дуже схожий на Lisp. А вирази Bash – це, здебільшого, S-вирази.

S-вираз – це список, у якому першим елементом є функція/команда/програма, а решта – аргументи.

```
echo Hello World  # функція echo друкує у термінал два свої аргументи,
                  # розділені одним пропуском
```

```
echo Hello     World  # дає той самий результат,
                      # число пропусків між масивами неважливе
```

У показаному вище прикладі є список із трьох елементів, `echo`, `Hello` і `World`, де функція `echo` повертає свої аргументи `Hello` та `World` розділені одним пропуском.

Якщо потрібно надрукувати «Hello World» з точною кількістю пропусків, ви повинні взяти пропуски в одинарні або подвійні лапки, щоб повідомити Bash, що це аргумент, а не роздільник.

```
echo "Hello     World"   # echo отримує один аргумент
echo  Hello "   " World  # echo отримує 3 аргументи -> той самий результат
echo Hello " " " " World # echo отримує 4 аргументи -> той самий результат
```

Отже, у Bash все є списком, де елементи розділені пропусками або символами табуляції. Елемент списку, залежно від контексту, може бути стрічкою, цілим числом або виразом, який розширюється до елементів.

```
A="Hello   World"   # Змінній A присвоюється стрічка "Hello   World".
echo $A             # Оператор $ розширює змінну в: echo Hello   World
B="echo   Hello  World"
$B                  # Змінна B розширюється у вираз: echo  Hello  World
                    # який оцінюється в CLI.
```

Можна оцінити результат функції за допомогою $(...).

```
echo $( echo "Hello     World" )
```

Внутрішнє `echo` розширюється до `Hello  World`, що, у свою чергу, як аргументи до зовнішнього `echo`, призводить до `"Hello World"`.

2. Оператори Bash

Оператори дозволяють виконувати операції над значеннями та змінними в Bash. Bash надає різні типи операторів для різних цілей:

- Арифметичні оператори.
- Оператори присвоєння.
- Оператори відношення.
- Логічні оператори.
- Порозрядні оператори.
- Тернарний оператор.

Вони комбінують цілочислові змінні та значення для отримання нового цілочислового результату. Арифметичні оператори дозволяють виконувати операції тільки над цілими числами. Нижче показано список усіх арифметичних операторів:

2.1. Арифметичні оператори

Оператор	Опис	Приклад
+	додавання	$a + $b
-	віднімання	$a - $b
*	множення	$a * $b
/	ділення	$a / $b
%	ділення за модулем (залишок)	$a % $b
**	піднесення до степеня	$a ** $b

2.2. Оператори присвоєння

Оператори присвоєння зберігають значення або результат виразу в змінній.

Оператор	Опис	Приклад
`=`	просте присвоєння	a=2
`+=`	додати і присвоїти	a+=2
`-=`	відняти і присвоїти	a-=2
`*=`	помножити і присвоїти	a*=2
`/=`	поділити і присвоїти	a/=2
`%=`	поділити за модулем і присвоїти	a%=2

2.3. Оператори відношень

Оператори відношень використовуються для оцінки умов порівняння цілих або стрічок

Оператор	Опис	Приклад
-eq	дорівнює	[$a -eq $b]
-ne	не дорівнює	[$a -ne $b]
-gt	більше ніж	[$a -gt $b]
-ge	більше або дорівнює	[$a -ge $b]
-lt	менше ніж	[$a -lt $b]
-le	менше або дорівнює	[$a -le $b]
`<`	менше ніж (стрічки)	["$a" < "$b"]
`<=`	менше або дорівнює (стрічки)	["$a" <= "$b"]
`>`	більше ніж (рядки)	["$a" > "$b"]
`>=`	більше або дорівнює (стрічки)	["$a" >= "$b"]
`==`	дорівнює	[$a ==- $b]
`!=`	не дорівнює	[$a != $b]

Оператори -eq, -ne, -gt, -ge, -lt і -le працюють з цілими числами, тоді як <, <=, > і >= працюють з порядком сортування стрічок. Вони звичайно використовуються в операторах if та циклах для керування потоком програми. Оператори рівності (==) і нерівності (!=) корисні для порівняння цілих чисел.

2.4. Логічні оператори

Логічні оператори використовуються для комбінування та заперечення виразів умов.

Оператор	Опис	Приклад
!	заперечення НЕ	! [$a -eq $b]
&&	логічне І	[$a -eq 5] && [$b -eq 10]
||	логічне АБО	[$a -eq 5] || [$b -eq 10]

Оператор NOT (!) перетворює умову true на false або умову false на true. Оператор І (&&) має значення істини, якщо обидва операнди є істинними, тоді як оператор АБО (||) має значення істини, якщо один із операндів є істинним. Вони дозволяють створювати складну логіку умов в сценаріях.

2.5. Порозрядні оператори

Оператор	Опис	Приклад
&	побітове І	$a & $b
|	побітове АБО	$a | $b
~	побітове НІ	$a ~ $b
^	побітове XOR	$a ^ $b
<<	зсув вліво	$a << 2
>>	зсув вправо	$a >> 2

Побітові оператори розглядають цілі числа як двійкові рядки та встановлюють, скасовують або перемикають біти в певних позиціях. Це дозволяє бітове маскування, перемикання та зміщення значень для прапорів і бінарних операцій низького рівня.

2.6. Тернарний оператор

Тернарний оператор допускає прості вирази умов. Він перевіряє задану умову та повертає вказаний результат залежно від того, чи оцінено умову як істинну чи хибну. Це забезпечує стислий спосіб призначення значень на основі умов. Синтаксис тернарного оператора

```
умова  ? результат-TRue :  результат-False
```

3. Арифметичні вирази

3.1. Команда let

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

$ let "var1=2**5";echo $var1
32
```

Одною командою `let` можна виконати декілька присвоєнь:

```
$ let "var1 = 6" "var2 = 2" "var3=var1+var2"; echo $var3
8
$ let "var1 = 6" "var2 = 2" "var3=var1-var2"; echo $var3
4
$ let "var1 = 6" "var2 = 2" "var3=var1*var2"; echo $var3
12
$ let "var1 = 6" "var2 = 2" "var3=var1/var2"; echo $var3
3
$ let "var1 = 7" "var2 = 2" "var3=var1%var2"; echo $var3
1
$ let "var1 = 10" "var2=var1++"; echo $var1 $var2
11 10
$ let "var1 = 8 >> 2"; echo $var1
2
$ let "var1 = 5" "var2 = 10" "var3 = var1|var2"; echo $var3
15
```

У виразі для обчислень команди `let` можна не використовувати знак розширення значення змінної `$`.

```
$ let ”SUM=SUM+5”
$ printf ”%d” ”$SUM”
15
$ let ”SUM=$SUM+5”
$ printf ”%d” ”$SUM”
20
```

Якщо змінна декларована як `integer` (ключ –i), то `let` команда виконується за замовчуванням:

```
$ declare –i SUM
$ SUM=SUM+5
$ printf “%d\n” $SUM
25
```

Приклад округлення до найближчого 10

```
$ declare -i COST=5234
$ COST=\(COST+5\)\10*10  # екранування круглих дужок
$ printf “%d\n” $COST
5230
```

Якщо змінна оголошена як стрічка (за замовчування всі змінні є стрічковими), то їй буде 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
```

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 виключальне АБО (XOR)
| – побітове АБО (OR)
&& – логічне І (AND)
|| – логічне АБО (OR)
expr ? expr1 : expr2 - тернарний оператор умови
=, *=, /=, %= – присвоєння
+=, -=, <<=, >>=, &=, ^=, |= – операції самопосилання

В команді `let` логічна істина (`true`) задається як 1, а логічна хиба (`false`) як 0. Любе значення відмінне від 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
```

3.2. Арифметичні вирази у круглих дужках ((...))

Команда `let` має синонім – подвійні круглі дужки ((…)). Вона використовується для того, щоб вбудовувати `let` вираз як параметр у іншу команду або для присвоєння значень змінним.

```
$ declare -i x=5
$ while (( x-- > 0 )); do printf "%d " "$x"; done
4 3 2 1 0
```

```
#!/bin/bash
val1=10
if (( $val1 ** 2 > 90 ))
then
    # let val2=$val1*$val1
   (( val2 = $val1 ** 2 ))
echo ”Квадрат $val1 дорівнює $val2”
fi
$ ./test
The Квадрат 10 дорівнює 100
```

В подвійних круглих дужках можуть використовуватися арифметичні цілочислові вирази і умови порівняння як із стандартними (як для команди команди test), так і з додатковими арифметичними операторам: **var++**, **var--**, **++var**, **--var**, **!** (логічна інверсія), **~** (побітова інверсія), ****** (експонента), **<<**, **>>** (зсув бітів вліво, вправо), **&** (бітове І), **|** (бітове АБО), **&&** (логічне І), **||** (логічне АБО).

За допомогою `$(( ... ))` можна оцінювати результат арифметичних виразів.

```
echo $(( 1 + 2 ))
echo $(( 1 + "2" ))      # Лапки не мають значення
echo $(( 2 ** 63 - 1 ))  # максимум 9223372036854775807
echo $(( 2 ** 63 ))      # мінімум -9223372036854775808
echo Hello $(( 8 / 4 )) # всередині  $((...)) цілі є цілими, а за межами вони є стрічки.
```

```
a=1;b=2
sum1=$(( 1+2 )); echo "sum1=$sum1"
(( a++ ))
sum2=$(( a + b )); echo "sum2=$sum2"
(( a++ ))
sum3=$(( $a + $b )); echo "sum3=$sum3"
(( a++ ))
let "sum4 = $(( a + b ))"; echo "sum4=$sum4"
(( a++ ))
let sum5=$((a+b)); echo "sum5=$sum5"
(( a++ ))
sum1=3
sum2=4
sum3=5
sum4=6
sum5=7
```

Однак немає підтримки чисел з плаваючою комою.

```
echo $(( 1 ** 3.14 )) # syntax error: invalid arithmetic operator
$ echo $(($var1 ** 2 > 30))
1
$ echo $((2**8))
256
```

Подвійні круглі дужки також використовуються для перетворення стрічок символів у арифметичні вирази `$((string)`:

```
$ sum=3+6
$ echo $sum
3+6

$ sum=$((3+6)); $ echo $sum
9
```

Значення змінним у Bash присвоюються як символи:

```
a=11;b=12
c=$a+$b; echo $c
11+12

c=$(( $a+$b )); echo $c
23
```

3.3. Одинарні круглі дужки (...)

Одинарні круглі дужки `(command)` використовуються для розширення команд Linux, тобто підставлення замість Linux команд їх стандартного виводу:

```
$ echo $(date)
```

```
Sun 25 Sep 2022 05:54:47 PM EEST
```

3.4. Команда expr для виконання арифметичних виразів

До додаткових можливостей виконання арифметичних виразів у Bash відносяться:

- використання команди `expr`;

- 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

Команда expr використовується для арифметичних обчислень в консолі

```
$ expr 6 + 3
9
$ expr 6 / 3
2
$ expr 6 \* 3
18
```

Для присвоєння значення обчисленого арифметичного виразу змінній, використовуються синтаксис розширення команд Linux у оболонці Bash `$(command)`:

```
#!/bin/bash
var1=10
var2=20
var3=$(expr $var2 / $var1)
echo Результат $var3
2
```

Екранування символу ”*” в команді `expr`:

```
$ expr 5 \* 2
10
```

3.5. Команда bc для арифметичних виразів з плаваючою крапкою

Команда `bc` (вбудований в Bash калькулятор) використовується для виконання арифметичних виразів з плаваючою крапкою. Точність виконання операцій задається змінною `scale`.

Синтаксис команди

```
bc [ -hlwsqv ] [long-options] [ file ... ]
```

-h, {- -help } : надрукувати дані про використання та вийти
-i, {- -interactive } : примусовий інтерактивний режим
-l, {- -mathlib } : визначає стандартну математичну бібліотеку
-w, {- -warn }: видавати попередження для розширень POSIX bc
-s, {- -standard } : обробляти саме мову POSIX bc
-q, {- -quiet } : не друкувати звичайне привітання GNU bc
-v, {- -version } : надрукувати номер версії та авторські права та вийти

Команда `bc` обробляє наступні оператори:
• арифметичні оператори;
• оператори збільшення або зменшення;
• оператори присвоювання;
• оператори порівняння або відношення;
• логічні або булеві оператори;
• математичні функції;
• вирази умов;
• вирази ітерацій.

Арифметичні оператори.

```
$ echo "12+5" | bc
17

$ bc -q
10 / 3
0

scale=4
10 / 3
3.3333
quit
$
```

Для використання калькулятора в сценарії використовується наступний синтаксис

```
variable=$(echo ”options; expression” | bc)
```

Приклад:

```
#!/bin/bash
var1=10
var2=3
var3=$(echo ”scale=4; $var1/$var2” | bc)
echo ”var3=$var3”

var3=3.3333
```

Оператори збільшення/зменшення.

```
$ echo "1var=10;var++" | bc
11
```

Оператори присвоєння.

```
$ echo "var=10;var^=2;var" | bc
100
```

```
$ echo "var=10;var^=2;var" | bc
100
```

Оператори порівняння або відношення.

```
$ echo "10>5" | bc
1
```

Логічні або булеві оператори.

```
$ echo "10 && 5" | bc
1
```

Математичні функції.

Команда `bc` підтримує наступні математичні функції:

• s (x): синус x, x у радіанах.

• c (x) : косинус x, x у радіанах.

• a (x) : арктангенс x, арктангенс повертає радіани.

• l (x) : натуральний логарифм x.

• e (x) : експоненціальна функція піднесення e до значення x.

• j (n,x) : функція Бесселя цілого порядку n від x.

• sqrt(x) : квадратний корінь із числа x. Якщо вираз від'ємний, генерується помилка виконання.

Підтримуються також додаткові функції:

• length(x) : повертає кількість цифр у x.

• read() : читає число зі стандартного потоку введення.

• scale(expression) : значенням функції масштабу є кількість цифр після коми у виразі.

• ibase і obase визначають базу перетворення для вхідних і вихідних чисел. За замовчуванням як для введення, так і для виведення є база 10.

• last (розширення) — це змінна, яка має значення останнього надрукованого числа.

```
$ echo "s($pi/3)" | bc -l
.86602540378443864675
```

```
$ echo "obase=2;15" | bc -l
1111
```

Вирази умов.

```
$ echo 'n=8;m=10;if(n>m) print "n>m" else print "n<m"' | bc -l
n<m
```

Вирази ітерацій.

```
$ echo "for(i=1; i<=10; i++) {i;}" | bc
1
...
10
```

```
$ echo "i=1;while(i<=10) {i; i+=1}" | bc
1
...
10
```

Вирази для виконання `bc` може вводити з файлу:

```
$ cat example.txt
2+5;
var = 10*3
var
print var
quit
```

```
$ bc example.txt
7
30
30
```

4. Вирази умов

4.1. Вирази з операторами if, if-then, if-then-else

В Bash оператор перевірки умови може мати різний синтаксис:

If command
then
  statements
fi
	If command
	then
	  statements1
	else
	  statements2
	  statements3
	fi
		If ((cmd1 && cmd2) || cmd3)
		then
		  statements1
		  statements2
		else
		  statements3
		fi
			If command1
			then
			  statements1
			elif command2
			then
			  statements2
			else
			  statements3
			fi

Оператори перевірки умови можна записати в один рядок, при цьому потрібно замінити символи нового рядка "\n" на символ ";".

```
If command;then statements;fi
If command;then statements1;else statements2;statements3;fi
If ((cmd1 && cmd2) || cmd3);then statements1;statements2;else statements3;fi
If command1;then statements1;elif statements2;else statements3;fi
```

В операторах перевірки умов використовуються значення істина (`true, 0`) і фальш (`false, не 0`):

```
$ true
$ printf "%d\n" "$?"
0
```

```
$ false
$ printf "%d\n" "$?"
1
```

Оператор перевірки умови `if` перевіряє код завершення любої команди Linux:

```
if grep -q Bash file; then echo "Файл містить, слово Bash";fi
```

Команда `test` є вбудованою у `Bash` і дозволяє перевіряти не *тільки код завершення команди*, але і *результат 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 хибу (1).

Команда може об'єднуватися з оператором `if-then`. Синтаксис команди `test`:

```
if test умова
then команда/команди
fi
```

Bash використовує як синонім для команди `test` одинарні квадратну дужку [(в реалізації права дужка] імітаційна). В нових реалізаціях `test` використовує подвійну квадратну дужку [[. Команда [є вбудованою у Bash.

```
$ type [
  - is a shell built-in

$ type ]
  not found


if [ умова ]
then команда/команди
fi

$ ls -ld /home
drwxr-xr-x 3 root root 4096 Feb 19  2023
$ test -d /home
$ echo $?
0

$ ls -ld /home
drwxr-xr-x 3 root root 4096 Feb 19  2023
$ [ -d /home ]
$ echo $?
0
```

Квадратні дужки є командою, то їх потрібно відділяти символом пропуску.

```
$ if [ $(echo TEST) ]; then echo CONDITION; fi
```

Оператор `if-then` перевіряє чи код завершення <команди/списку команд> 0 (істина), і якщо так, то виконує одну або більше команд, які слідують за словом `then`.

`if command1; … commandN`
`then`
`  commands`
`fi`
	`if command; … commandN; then`
	`  commands`
	`fi`

```
$ if date
$ then
$   echo «це працює»
$ fi
Це працює

$ if date; then echo «це працює»; fi
Це працює
```

Для оброблення альтернативної команди використовується оператор `if-then-else`:

if *command*
then
  commands
else
  commands
fi
	if *command;* then
	 commands
	else *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
```

Оператор `if-then-else` може мати вкладені перевірки умов.

if *command1*
then
  command set 1
elif *command2*
then
  command set 2
elif *command3*
then
  command set 3
elif *command4*
then
  command set 4
fi
	if *command1;* then
	  command set 1
	elif *command2;* then
	  command set 2
	elif *command3;* then
	  command set 3
	elif *command4* ; then
	  command set 4
	fi

4.2. Оцінювання умов порівняння командою `test, [ ... ]`

Команда `test` і її синонім [...] використовується для оцінювання умов порівняння

```
$ test  5 -eq 6; echo $?
1
```

```
$ [ 5 -eq 6 ]; echo $?
1
```

```
$ test 100 -lt 99 && echo "Yes." || echo "No."
No.
```

```
$ [ 100 -lt 99 ] && echo "Yes." || echo "No."
No.
```

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 ]; then echo ”gt”; fi
$ test 100 -lt 99 && echo "Так" || echo "Ні"
```

Зарезервована змінна `$?` містить код завершення команди (або програми)

```
$ [ 1 -eq 2 ]; echo $?
1
```

• **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).

`[ st1 = str1 ]` – символьні стрічки однакові
`[ str1 != str 2 ]` – символьні стрічки не однакові
`[ st1 \< str1 ]` – чи `str1` менша лексикографічно від `str2`. `\<` - екранування зарезервованого символу
`[ st1 \> str1 ]` – чи `str1` більша лексикографічно від `str2`. `\>` - екранування зарезервованого символу
`[ -n str ]`, `[ str ]` – не нульова довжина символьної стрічки
`[ -z str ]` – нульова довжина символьної стрічки

```
$[ 't' \> 'T' ]; echo $?
0
```
– t_{ASCII}=116, T_{ASCII}=84

```
#!/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 файлу *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 порожній
-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
```

4.3. Подвійні квадратні дужки

Варіант нової команди `test` із синтаксисом `[[ вираз ]]` має розширені можливості. Всередині цієї конструкції не проводиться додаткова інтерпретація імен файлів, а аргументи не розбиваються на окремі слова, що забезпечує розширені умови порівняння символьних стрічок, наприклад порівняння з використанням регулярних виразів (шаблонів). Приклад перевірки значення змінної середовища `$USER`, чи вона починається з букви “`R`”:

```
if [[ $USER == R* ]] …
```

```
#!/bin/bash
echo "Введіть username"
read username
echo "Введіть password"
read password
```

Складна умова порівняння символьних стрічок

```
if [[ ( $username == "admin" && $password == "secret" ) ]]; then
echo "valid user"
else
echo "invalid user"
fi
```

5. Оператор case

Для вибору значення однієї змінної з багатьох можливих значень (замість інструкції `if-then-else`) використовується оператор `case`. Оператор **`case`** підтримує багатозначний вибір для однієї змінної. Синтаксис оператора `case`:

case expression in
  pattern_1)
  statements
  ;;
  pattern_2)
  statements
  ;;
  pattern_n)
  statements
  ;;
  *)
  statements
  ;;
	case *variable* in
	pattern1 | *pattern2*) *commands1*;;
	pattern3) *commands2*;;
	**) default commands*;;
	esac

```
#!/bin/bash
DEPARTMENT=("Комп’ютерні науки”, ”Електронна інженерія”, ”Електрична інженерія”, ”Інформаційні технології”)
for value in "${DEPARTMENT[@]}"
do
echo -n "Your DEPARTMENT is "
case $DEPARTMENT in
  "Комп’ютерні науки")
    echo -n "Комп’ютерні науки"
    ;;
   "Електронної інженерії" | "Електричної інженерії")
    echo -n "Електронної інженерії і Електричної інженерії"
    ;;
  "Інформаційні технології")
    echo -n "Інформаційні технології"
    ;;
  *)
    echo -n "Помилка"
    ;;
esac
done

#!/bin/bash
USER=”Петро”
case $USER in
Іван)
echo ”Привіт $USER”;;
Петро)
echo ”$USER у вас обмежений доступ”;;
*)
echo ”Вхід заборонений”;;
esac

#!/bin/bash
echo -n "Чи Ви студент? [yes or no]: "
read response
case $response in
        "Y" | "y" | "YES" | "Yes" | "yes")
                echo -n "Так, я студент."  ;;
        "N" | "n" | "No" | "NO" | "no" | "nO")
```

```
                echo -n "Hi, я не студент."  ;;
        *) echo -n "Помилка"  ;;
esac
```

```
#!/bin/bash
for arg in "$@"
do
index=$(echo $arg | cut -f1 -d=)
val=$(echo   $arg | cut -f2 -d=)
case $index in
X) x=$val;;

Y) y=$val;;

*)
esac
done
((result=x+y))
echo "X+Y=$result"
```

```
$ bash command_line_arg.bash X=10 Y=20
X+Y=30
```

6. Оператори циклів

Оператор **for** дозволяє організувати цикл, який перебирає серію значень:

for var in 1 ... N
do
  command
done
	for var in file1 ... fileN
	do
	  command
	done
		for output in $(command)
		do
		  command
		done

```
for i in {1..5}; do echo $i; done
```

Виведення вмісту поточного каталогу:

```
$ for i in $(ls -l); do echo $i; 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 i in $(cat $my_file)
do
  echo $i
done
```

Змінна середовища $IFS містить значення розділювача полів у стрічках, за замовчуванням пропуск. Це значення можна поміняти на інше або інші значення, наприклад `IFS=":;."` встановить три розділювачі полів - ”:”,”;”,”.”.

```
#!/bin/bash
# встановлення розділювача ”\n”.
file=”/etc/passwd”
IFS=$’\n’
for var in $(cat $file)
do
  echo $var
done
```

```
#!/bin/bash
# встановлення трьох розділювачів полів
# зберегти поточне значення розділювача полів
IFS_OLD=$IFS
# встановити нові значення
IFS=”:;.”
file="states1"
echo "Штати"
for state in $(cat $file)
do
echo "$state"
done
# відновлення значення
IFS=$IFS_OLD
```

Цикли у стилі мови програмування С:

```
for ((i=1;i<=3;i++))
do
  echo ”Цикл у стилі С: $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”
```

Оператор `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 використанням змінної

```
#!/bin/bash
x=0;
while [ ”$x” -le 10 ]
do
  echo ”x=$x”
  x=$(($x++))
done
```

Оператор `until умова` використовується для організації циклів з перевіркою умови. *Цикл виконується поки* `умова` *хибна*. Так в прикладі цикл зупиниться, коли величина `x` досягне значення 10.

```
#!/bin/bash
x=10
until [ ”$x” -eq 0 ]
do
   echo ”x=$x”
   x=$(($x-1))
done
```

```
x=10; until [ "$x" -eq 2 ]; do echo "x=$x"; x=$((x - 1)); done
```

Для виходу із циклу використовується інструкція **`break [n]`**, де `n` – номер вкладеності циклу: 1 (за замовчуванням) самий внутрішній вкладений цикл), 2 – наступний внутрішній вкладений цикл, ..., n – зовнішній охоплюючий цикл.

Для пропущення поточної ітерації і продовження циклу використовується інструкція **`continue [n]`**, де `n` – номер вкладеності циклу.

```
#!/bin/bash
for i in 1 2
do
for j in a b
do
for k in 4 5
do
echo $i $j $k
if [ $k -eq 5 ]; then break n; fi
done
done
done
```

Результат виконання для

n=1	n=2	n=3
1 a 4	1 a 4	1 a 4
1 a 5	1 a 5	1 a 5
1 b 4	2 a 4	
1 b 5	2 a 5	
2 a 4		
2 a 5		
2 b 4		
2 b 5		

Перенаправлення даних із циклів:
- перенаправлення у файл `do … done > out.txt`
- перенаправлення в іншу команду `do … done | sort`

7. Маніпуляції стрічками

Інтерпретація послідовності керуючих символів $'…'

Наступні послідовності символів можуть бути інтерпретовані у символьних стрічках як керуючі символи:

`\a - Alert (bell), \b - Backspace, \c - control character C, \e - Escape character, \f - Form feed, \n - New line, \r - Carriage return, \t - Horizontal tab, \v - Vertical tab, \\ - Literal backslash, \' - single quote, \nnn - ASCII octal, \xnnn - ASCII hexadecimal`

```
$ printf "%s\n" $'line 1\nline 2\n'
line 1
line 2
```

```
$ printf "%s\n" $"line 1\nline 2\n'
line 1\nline 2\n
Подвійні лапки відміняють дію керуючих символів
```

Перевірка наявності змінної

```
x=5
$ printf "%s\n" "${x:?Змінна не оголошена}"
5
$ printf "%s\n" "${y:?Змінна не оголошена}"
bash: y: Змінна не оголошена
```

Довжина символьної стрічки

`${#string}`
`expr length $string`
`expr "$string" : '.*'`

```
string=abcABC123ABCabc
echo ${#string}                 # 15
echo $(expr length $string)     # 15
echo $(expr "$string" : '.*')   # 15
```

```
$ COMPANY="Nightlight Inc."
$ printf "%s\n" "${#COMPANY}"
15
```

Довжина співпадіння підстрічки (регулярний вираз) від початку стрічки

```
expr match "$string" '$substring'
expr "$string" : '$substring'
```

```
string=abcABC123ABCabc
#      |------|
#      12345678
echo $(expr match "$string" 'abc[A-Z]*.2')  # 8
echo $(expr "$string" : 'abc[A-Z]*.2')      # 8
```

Індекс співпадіння підстрічки

```
expr index $string $substring
```

```
string=abcABC123ABCabc
#      123456 ...
echo $(expr index "$string" C12)            # 6
echo $(expr index "$string" xAz)            # 4
```

Видобування підстрічки у стрічці із заданого індексу [, заданої довжини]

```
${string:position}
${string:position:length}
```

```
string=abcABC123ABCabc
#      0123456789.....
#      0-початок індексування
echo ${string:0}                            # abcABC123ABCabc
echo ${string:1}                            # bcABC123ABCabc
echo ${string:7}                            # 23ABCabc
echo ${string:7:3}                          # 23A
hello="Привіт світ"
```

```
$ printf "%s\n" "${hello:3}"
віт світ
$ printf "%s\n" "${hello:7:5}"
світ
```

Вилучення підстрічки у заданій стрічці з її початку

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.

```
${string#substring}   # найкоротше співпадіння
${string##substring}  # найдовше співпадіння

string=abcABC123ABCabc
#      |----|          shortest
#      |----------|    longest
echo ${string#a*C}      # 123ABCabc (правіше від співпадіння)
echo ${string##a*C}     # abc (правіше від співпадіння
X='a*C'  # підстрічку можна параметризувати
echo ${string#$X}       # 123ABCabc
echo ${string##$X}      # abc
```

Вилучення підстрічки у заданій стрічці з її кінця

```
${string%substring}   # найкоротше співпадіння (лівіше від співпадіння)
${string%%substring}  # найдовше співпадіння (лівіше від співпадіння)
```

```
# Перейменувати всі файли in $PWD із суфікса "TXT" на суфікс "txt"
# Наприклад, "file1.TXT" на "file1.txt"
SUFF=TXT
suff=txt
for i in $(ls *.$SUFF)
do
  mv -f $i ${i%.$SUFF}.$suff
done

string=abcABC123ABCabc
#                    ||     найкоротше
#       |------------|     найдовше
echo ${string%b*c}      # abcABC123ABCa
echo ${string%%b*c}     # a

$ 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 підстрічок у стрічці

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.

${string/substring/replacement}  # заміна першого співпадіння
${string//substring/replacement} # заміна всіх співпадінь
${string/#substring/replacement} # заміна співпадіння на початку стрічки
${string/%substring/replacement} # заміна співпадіння в кінці стрічки

string=abcABC123ABCabc
echo ${string/abc/xyz}       # xyzABC123ABCabc
echo ${string//abc/xyz}      # xyzABC123ABCxyz
echo ”$string”               # abcABC123ABCabc – сама стрічка без змін

                              # The string itself is not altered!
# параметризація заміни
match=abc
repl=000
echo ${string/$match/$repl}  # 000ABC123ABCabc
echo ${string//$match/$repl} # 000ABC123ABC000
```

```
# якщо $replacement не задано
echo ${string/abc}           # ABC123ABCabc
echo ${string//abc}          # ABC123ABC
```

Запитання.

1. Типи даних Bash.
2. Оператори Bash.
3. Арифметичні вирази.
4. Вирази умов.
5. Оператор case.
6. Оператори циклів.
7. Маніпуляції стрічками.

7. ПЕРЕНАПРАВЛЕННЯ ПОТОКІВ. КАНАЛИ

Мета. Вивчення перенаправлення потоків введення-виведення даних та каналів у 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. Команда exec
   3.1. Використання команди `exec` для заміни процесу
   3.2. Використання команди exec для заміни сесії поточної оболонки
   3.3. Виклики програм за допомогою exec у Bash сценаріях
   3.4. Створення порожнього середовища
   3.5. Виконання з командою find
   3.6. Перенаправлення дескрипторів файлів у файл реєстрації
4. Тимчасове і постійне перенаправлення
5. Створення власних перенаправлень виведення
6. Створення дескрипторів для читання-записування
7. Закриття файлових дескрипторів
8. Інформація про файлові дескриптори
9. Подавлення виведення
10. Використання тимчасових файлів і каталогів
11. Анонімні і іменовані канали
    11.1. Змінна Bash PIPESTATUS
    11.2. Іменований канал
    11.3. Перенаправлення виводу stdout множини команд
12. Співпрограми

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 STDOUT на дескриптор іншого файлу.

```
> cat > out # приймає введення з консолі, виведення у файл out
111  # введено
222  # введено
<CTRL+D>  # завершення введення
```

Використовуючи символ ″<″ можна переназначити файловий дескриптор введення консолі на дескриптор іншого файлу.

```
> cat < file   # введення даних з файлу і виведення даних на екран
111
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)
$ one=1;two=2
$ printf ”%s\n” ”digits $one $two”
digits 1 2
$ printf ”%s %d %d\n” ”digits” $one $two
digits 1 2
```

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 признаки початку даних
>дані
…
>дані
>символи признаки кінця даних (мають співпадати із символами початку)
```

Приклад (EOF може бути люба стрічка):

```
>wc <<EOF
>1
>2
>3
>EOF
3 3 6
```

<<< - направлення стрічки з правої сторони на STDIN команди зліва:

```
$ cat <<<  'Hi there'
Hi there
```

Для передачі даних з виходу однієї команди на вхід іншої використовується канал (або конвеєр) | (англ. pipe).

```
command1 | command2
```

Так, виконання двох команд можна замінити одною командою:

```
>rpm –qa > rpm.list
>sort < rpm.list
>rpm –qa | sort > rpm.list
```

Регістро незалежний пошук стрічки “error” у файлі log, перенаправлення результату на вхід команди `wc`, яка підраховує кількість стрічок:

```
$ grep –i ”error” ./log | wc
```

Команди, які виконуються в каналі, виконуються у підоболонці. Всі команди конвеєра виконуються в окремих процесах, створених викликом `clone()`.

```
a=”aaa”; b=”bbb”
$ echo "Hello world!" | (read a b; echo $a; echo $b)
Hello
world!
```

```
$ a=”aaa”; b=”bbb”
$ echo ”Hello world!” | read a b | (echo $a; echo $b)
aaa
bbb
```

Так, очікується значення змінних a="Hello", b="world!", а в дійсності вони залишаться a="aaa", b="bbb". Взагалі любі зміни значень змінних в конвеєрі залишить ці змінні без змін за його межами. :

```
$ filefound=0
$ find . type f –size +10k |
  while true
    do
      read f
      echo "$f більше 10k"
      filefound=1
      break
    done
$ echo $filefound
```

Навіть якщо буде знайдений файл більше 10 Кб, прапор filefound все одно буде мати значення 0.

Потрібно перенести всю логіку обробки значень змінних в той же підпроцес у конвеєр:

```
$ echo "one two" | (read a; echo a;)
one
```

Змінений приклад з find:

```
$ filefound=0
$ for f in $(find . type f –size +10k)
    do
      read f
      echo "$f більше 10k"
      filefound=1
      break
    done
$ echo $filefound
```

Перенаправлення стандартних файлових дескрипторів виведення у файл:

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
```

Після виконання команди перенаправлення втрачаються.

Для створення реєстраційних файлів (log files) використовується команда `tee` (трійник, розгалуження). Вона дозволяє направити дані із STDIN у STDOUT (за замовчуванням) і у вказаний файл за один раз

```
tee filename
$echo ”Повідомлення” | tee file.log # виведення в STDOUT і оновлення файлу file.log
$echo ”Повідомлення” | tee >> file.log # виведення в STDOUT і добавлення у файл file.log
```

Команда `tee` також використовується з командою створення каналу для перенаправлення виходу любої команди.

```
who | tee testfile
```

2. Команда exec

Команда Linux exec виконує Shell команду без створення нового процесу. Натомість вона замінює поточну відкриту оболонку Shell. Залежно від використання команда exec має різну поведінку та випадки використання.

Синтаксис команди exec

```
exec [options] [command [arguments]] [redirection]
```

Опції:

-c – виконання команди в порожньому середовищі;

-l – передати "-" як нульовий аргумент;

-a [name] – передати [name] як нульовий аргумент.

3.1. Використання команди `exec` для заміни процесу

1. Вивести список процесів

```
$ ps
PID TTY          TIME CMD
6163 pts/1    00:00:00 bash
6238 pts/1    00:00:00 ps
```

2. Вивести PID поточного процесу

```
$ echo $$
6163
```

3. Запустити команду `exec` і передати їй команду `sleep 100`

```
$ exec sleep 100
```

4. В іншому терміналі із списку процесів вибрати процес `sleep`

```
$ ps -ae | grep sleep
6163 pts/1 00:00:00 sleep
```

PID процесу `sleep` є такий самий, як у оболонки Bash, це показує, що команда `exec` замінила процес оболонки Bash.

3.2. Використання команди exec для заміни сесії поточної оболонки

1. Вивести PID поточної оболонки

```
$ echo $$
6494
```

2. Командою exec перемкнутися на процес Bourne оболонки (sh)

```
$ exec sh
sh-4.4$
```

3. В іншому терміналі із списку процесів вибрати процес `sh`

```
$ ps -ae | grep "sh\b"
6494 pts/1 00:00:00 sh

6604 pts/2 99:00:00 bash
```

Команда замінила Bash процес (bash) на Bourne процес (sh). Вихід із Bourne оболонки закриває термінал.

3.3. Виклики програм за допомогою exec у Bash сценаріях

```
# test.sh
#!/bin/bash
while
do
 echo "1. Nano"
 echo "2. Vim"
 echo "3. Exit"
 read input
 case "$input" in
       1) exec nano ;;
       2) exec vim ;;
       3) break
 esac
done
```

Запуск на виконання

Після виконання сценарію термінал закривається.

```
$ . test.sh
```

Після виконання сценарію термінал не закривається.

```
$ bash test.sh
```

3.4. Створення порожнього середовища

```
$ exrc -c printenv
```

3.5. Виконання з командою find

Команда `find` у Linux має команду `exec` як опцію для виконання дії над виявленим вмістом. Наприклад, рядок нижче виконує команду `chmod` для результатів команди `find`:

```
$ find ~ -name "test.log" -exec chmod +x '{}' \;
```

3.6. Перенаправлення дескрипторів файлів у файл реєстрації

```
#!/bin/bash
# Створення файлу test.log
```

```
touch test.log
# Збереження test.log у змінну log_file
log_file="test.log"
# Перенаправлення stdout у $log_file
exec 1>>$log_file
# Перенаправлення stderr туди ж де і stdout
exec 2>&1
echo "Рядок 1 добавлено у файл test.log"
echo "Рядок 2 добавлено у файл test.log"
eho "Рядок 3 має помилку і замість нього буде записано повідомлення stderr"
```

4. Тимчасове і постійне перенаправлення

Файлові дескриптори можуть бути перенаправлені:
– тимчасово для одноразового виконання команди;
– постійно для всіх команд сценарію.

Перенаправлення виведення.

Для *тимчасового* перенаправлення файлового дескриптора перед його номером ставиться символ амперсанду `&`:

```
# 1.sh
#!/bin/bash
echo ”Це помилка” >&2
echo ”Це нормальне виведення”
```

```
$ ./1.sh
Це помилка
Це нормальне виведення
```

Якщо запустити сценарій, то обидві стрічки попадуть на екран, так як за замовчуванням STDOUT і STDERR виводяться на екран.

Запуск сценарію для виведення stderr у файл

```
$ ./1.sh 2> file.err
Це нормальне виведення
$ cat file.err
Це помилка
```

Для *постійного* перенаправлення файлового дескриптора використовується команда `exec`, яка запускає нову оболонку `Bash` і перенаправляє файловий дескриптор на час дії сценарію:

```
exec 1> testout  # перенаправлення стандартного виведення у файл
exec 2> err      # перенаправлення помилок у файл
```

Команда `exec` може розміщуватися в різних місцях сценарію:

```
# 2.sh
#!/bin/bash
exec 2> err
echo "Початок сценарію > stdout"
echo "Виведення помилки у stderr"
ls xxx
echo "Постійне перенаправлення stdout у файл out"
exec 1> out
echo "Виведення 1 у файл out"
echo "Тимчасове перенаправлення stderr у файл file.err" >&2
```

```
echo "Виведення 2 у файл out"
```

```
$ ./2.sh
Початок сценарію > stdout
Виведення помилки у stderr
```

```
$ cat out
Виведення 1 у файл out
Виведення 2 у файл out
```

```
$ cat file.err
Тимчасове перенаправлення stderr у файл file.err
```

Перенаправлення введення.

Можна постійно перенаправити стандартний потік введення STDIN з клавіатури на введення з файлу командою exec:

```
exec 0< myfile
```

Ця команда вказує оболонці, що джерелом даних має стати файл `myfile`, а не звичайний `stdin`.

```
# 3.sh
#!/bin/bash
exec 0< myfile
count=1
while read line
do
echo "Рядок $count: $line"
count=$(( $count + 1 ))
done
```

```
exec 0< file
while read line
do
   echo $line
done
```

```
$ ./3.sh
Рядок 1: 1111
Рядок 2: 2222
Рядок 3: 3333
Рядок 4: 4444
```

Команду `exec` можна використовувати багаторазово і перенаправляти вихід для різних файлових дескрипторів. Після перенаправлення можна завжди повернутися до стандартних дескрипторів.

5. Створення власних перенаправлень виведення

Оболонка Bash дозволяє відкриття 9-ти файлових дескрипторів. Дескриптори з третього по восьмий можуть використовуватися для перенаправлення як введення, так і виведення.

Альтернативні файлові дескриптори можна призначити стандартним, а стандартні - альтернативним.

```
exec 1>&6
echo "Призначити stdout в 6" >&1
…
```

```
exec 3>&1
echo "Призначити дескриптора 3 в stdout" >&3
…
```

Призначити дескриптор 3 для виведення даних у файл:

```
# 4.sh
#!/bin/bash
exec 3> myfile
echo "Виведення 1 на екран"
echo "Виведення у файл" >&3
echo "Виведення 2 на екран"
```

```
$ ./4.sh
Виведення 1 на екран
Виведення 2 на екран
```

```
$ cat myfile
Виведення у файл
```

Аналогічно можна призначити STDIN іншому файловому дескриптору, а потім відновити стандартне значення STDIN.

```
exec 6<&0
exec 0< testfile
read line
echo $line
exec 0<&6
```

Приклад переназначення STDIN:

```
# 5.sh
#!/bin/bash
exec 6<&0
exec 0< myfile
count=1
while read line
do
echo "Рядок $count: $line"
count=$(( count + 1 ))
done
exec 0<&6
read -p "Завершити ? " answer
case $answer in
y) echo "До побачення";;
n) echo "Кінець файлу";;
esac
```

```
$ ./5.sh
Рядок 1: 1111
Рядок 2: 2222
Рядок 3: 3333
Рядок 4: 4444
Рядок 5:
Завершити ? y
До побачення
```

У цьому прикладі дескриптор файлу 6 використовується для зберігання посилання на STDIN. Потім було зроблено переназначення і джерелом даних став файл. Після цього вхідні

дані для команди read поступали із переназначеного STDIN, тобто із файлу. Після читання файлу STDIN повертаються в початкове положення, переназначивши його в дескриптор 6.

6. Створення дескрипторів для читання-записування

Файлові дескриптори з можливістю читання-записування створюються командою `exec <>` з відповідним номером файлового дескриптора:

```
exec 3<> file
read line <&3
echo ”Test” >&3
```

7. 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
```

8. Інформація про файлові дескриптори

Отримати інформацію про відкриті файлові дескриптори всіх процесів Linux дозволяє команда `/usr/bin/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
```

Приклади:

```
>lsof –a –p $$ -d 0,1,2
COMMAND  PID  USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash    2602 inter    0u   CHR  136,1      0t0    4 /dev/pts/1
bash    2602 inter    1u   CHR  136,1      0t0    4 /dev/pts/1
bash    2602 inter    2u   CHR  136,1      0t0    4 /dev/pts/1
```

Типи файлів, зв’язаних з STDIN, STDOUT, STDERR – CHR (character mode, символьний режим). Так як всі вони вказують на термінал, то ім’я файлу відповідає імені пристрою, призначеного терміналу. Всі три стандартні файли доступні для читання і записування.

Наступний сценарій відкриває два дескриптори (3 і 6) для виведення і один – для введення (7).

```
#!/bin/bash
exec 3> myfile1
```

```
exec 6> myfile2
exec 7< myfile3
lsof -a -p $$ -d 0,1,2,3,6,7

COMMAND  PID  USER   FD   TYPE DEVICE SIZE/OFF      NODE NAME
7.sh    4204 inter    0u   CHR  136,1      0t0         4 /dev/pts/1
7.sh    4204 inter    1u   CHR  136,1      0t0         4 /dev/pts/1
7.sh    4204 inter    2u   CHR  136,1      0t0         4 /dev/pts/1
7.sh    4204 inter    3w   REG    8,6        0 134501148 /home/inter/myfile1
7.sh    4204 inter    6w   REG    8,6        0 134501149 /home/inter/myfile2
7.sh    4204 inter    7r   REG    8,6        0 134501150 /home/inter/myfile3
```

9. Подавлення виведення

Іноді потрібно зробити так, щоб команди в сценарії, який може виконуватися як фоновий процес, нічого не виводили на екран. Для цього можна перенаправити виведення в /dev/null.

```
$ ls –al file1 file2 2> /dev/null
```

Використовуючи такий підхід можна очистити файл, не вилучаючи його.

```
$ cat /dev/null > myfile
```

10. 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.

Тимчасові файли можна створювати командою `touch` або `mktemp`:

```
$ touch /tmp/filename
```

Команда `mktemp` з відповідними ключами створює тимчасові файли і каталоги. 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.xxxxxxxxxx   #  10 символів після крапки
```

Команда замінює шаблон `xx…xx` у імені файлу унікальним іменем. Звичайно xx..x замінюється комбінацією поточного номера процесу і випадкових букв.

При запуску команди без параметрів розширення файлу `tmp` має унікальне випадкове значення.

```
$ mktemp
/tmp/tmp.yTfJX3516c
```

Приклад створення тимчасового файлу і записування у нього даних:

```
tempfile =$(mktemp /tmp/tmp.a1b2c3d4e5)
exec 3>$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
```

11. Анонімні і іменовані канали

Під час роботи з інтерфейсом командного рядка Linux прийнято перенаправляти вихідні дані одної програми на використання як вхідні дані для іншої програми. Для цього використовуються канали.

Канал є важливим механізмом у системах на основі Unix/Linux, який дозволяє передавати дані від одного процесу до іншого, не зберігаючи нічого на диску. У Linux є два типи каналів:

• анонімні або безіменні канали ('|');

• FIFO або іменовані канали.

Анонімний канал використовується шляхом поєднання команд, розділених символом вертикальної лінії ' | '. Його часто називають конвеєром і *кожна оболонка визначає його поведінку*.

Оболонка виконує кожну команду в окремому процесі, який працює у фоновому режимі, починаючи з крайньої лівої команди. Тоді стандартний вихід команди з лівого боку з’єднується зі стандартним входом команди з правого боку. Це забезпечує односпрямованість потоку. Цей механізм діє до завершення всіх процесів у конвеєрі.

Такі оболонки, як Bash і Zsh, використовують маркер “ |& ” для позначення конвеєра, що з’єднує як стандартний вихід, так і стандартну помилку команди зліва зі стандартним входом команди з правого боку.

Припустимо, що хочемо використати команду `netstat` , щоб побачити, які процеси запущені за допомогою `localhost` і фільтрувати за допомогою утиліти `grep` :

```
$ netstat –tlpn | grep 127.0.0.1
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
tcp 0 0 127.0.0.1:3306 0.0.0.0:* LISTEN -
```

У цьому прикладі вихідних даних видно, що `stdout` і `stderr` від `netstat` не залежать від фільтру. Тепер об’єднаємо `stderr` із `stdout` і передамо його на `stdin` команди `grep` :

```
$ netstat –tlpn |& grep 127.0.0.1
tcp 0 0 127.0.0.1:3306 0.0.0.0:* LISTEN -
```

Тепер попереджувальне повідомлення приховано.

11.1. Змінна Bash PIPESTATUS

Bash має змінну під назвою `PIPESTATUS`, яка містить список статусів виходу з процесів у останньому виконаному конвеєрі:

```
$ exit 1 | exit 2 | exit 3 | exit 4 | exit 5
$ echo ${PIPESTATUS[@]}
1 2 3 4 5
```

Статус повернення виконання всього конвеєра залежатиме від статусу змінної `pipefail`. Якщо цю змінну встановлено, статус повернення конвеєра буде статусом виходу крайньої правої команди з ненульовим статусом або матиме нуль, якщо всі команди завершаться успішно:

```
$ set -o pipefail
$ exit 1 | exit 2 | exit 3| exit 4 | exit 0
$ echo $?
4
```

Якщо параметр `pipefail` вимкнено, статус повернення каналу буде статусом виходу останньої команди:

```
$ set +o pipefail
$ exit 1 | exit 2 | exit 3| exit 4 | exit 0
$ echo $?
0
```

Bash також має опцію `lastpipe`, яка наказує оболонці виконати останню команду на передньому плані поточного середовища.

11.2. Іменований канал

Іменований канал (також званий іменованим FIFO або просто FIFO) – це канал, точкою доступу якого є файл, що зберігається у файловій системі. Відкриваючи цей файл для читання, процес отримує доступ до кінця каналу для читання. Відкриваючи файл для запису, процес отримує доступ до кінця каналу для запису. Якщо процес відкриває файл для читання, він блокується, доки інший процес не відкриє файл для запису. Аналогічна ситуація і при відкритті файлу для запису.

FIFO має ті самі характеристики, що й будь-який інший файл. Наприклад, він має право власності, дозволи та метадані.

Іншою важливою особливістю FIFO є те, що він забезпечує двонаправлений зв'язок . У Linux можна створити іменований FIFO за допомогою команд `mkfifo` або `mknode`:

```
$ mkfifo pipe1; mknode pipe2 p
$ ls -l
prw-r--r-- 1 user user 0 Oct 14 21:17 pipe1
prw-r--r-- 1 user user 0 Oct 14 21:17 pipe2
```

Як видно, тип файлів FIFO позначається літерою «`p`».

Приклад роботи іменованого каналу `pipe1`. В одній віртуальній консолі1 ввести:

```
$ ls -al > pipe1
```

В іншій віртуальній консолі2 ввести:

```
$ cat < pipe1
```

Команда, виконана у першій консолі, відображається на другій консолі. Зауважимо, що порядок виконання команд `ls` і `cat` не має значення. Після запуску першої команди її процес блокується. Це відбувається тому, що інший кінець каналу ще не підключено, і тому ядро призупиняє перший процес, доки другий процес не відкриє канал.

Канал може бути відкритий як для читання, так і для запису. Можна запустити дві оболонки і організувати між ними обмін повідомлення через FIFO.

Оболонка 1

```
$ echo "Оболонка 1" > pipe1
```

Оболонка 2

```
$ cat < pipe1 > pipe2
```

Оболонка 1

```
cat < pipe2
Оболонка 1
```

У канал можна направляти результати виконання декількох команд з використанням круглих або фігурних дужок. Якщо команди взяти у круглі дужки, то оболонка створює підоболонку, яка інтерпретує команди у дужках. Оскільки зовнішня оболонка виконує лише одну «команду», вихід повного набору команд можна перенаправляти як одиницю. Наприклад, команда:

```
$(echo "Оболонка1"; pwd) > pipe1
```

записує результат виконання команд `echo` і `pwd` у канал `pipe1`.

Якщо команди взяти у фігурні дужки, то всі команди виконуються в одній оболонці.

```
${ echo echo "Оболонка1"; pwd; } > pipe1
```

Іменовані та анонімні канали можна використовувати разом. Можна створити зворотну оболонку, поєднуючи FIFO і анонімні канали.

Використаємо утиліту `nc` для створення програми клієнт/сервер, у якій «серверна» сторона надасть свою оболонку, а «клієнтська » матиме доступ до неї.

Спочатку потрібно встановимо пакет `netcat-openbsd`:

```
$ sudo apt install netcat-openbsd
```

Далі створюється FIFO під назвою `fifo_reverse`,

```
$ mkfifo fifo_reverse
```

Тоді потрібно зайти у систему як два користувачі і запустити консолі, одна з них діятиме як «клієнт» (скажімо, «user1»), а інша, як «сервер» (скажімо, «user2»).

Перемикання між користувачами

```
$ whoami
$ user1
$ sudo - user2 user2_psw
$ whoami
$ user2
```

Конвеєр запустимо в оболонці user2:

```
$ user2$ bash -i < fifo_reverse |& nc -l 127.0.0.1 1234 > fifo_reverse
```

У цьому однорядковому коді оболонка читає вміст FIFO та передає його в інтерактивну оболонку Bash.

Далі як `stdout`, так і `stderr` інтерактивної оболонки буде передано команді `nc`, яка буде слухати порт 1234 адреси 127.0.0.1.

Тепер, коли «клієнт» успішно встановлює з'єднання, `nc` запише отримане в FIFO, а інтерактивна оболонка зможе виконати отримане.

Тепер в оболонці user1, виконується команда `nc`:

```
$ user1$ nc 127.0.0.1 1234
user2$
```

В результаті отримано рядок запрошення оболонки user2, але використовуючи оболонку user1, яка поєднує анонімні та іменовані канали.

11.3. Перенаправлення виводу `stdout` множини команд

Перенаправлення виводу `stdout` одної команди на `stdin` іншої команди є потужною технікою. Однак бувають випадки коли потрібно перенаправити вивід `stdout` декількох команд. У цьому випадку використовується техніка заміни процесу (англ. process substitution). Заміна процесу направляє виводи декількох команд на вхід одної команди. Позначення цього механізму для передачі виводу списку команд на стандартний вхід команди:

```
<(список команд)
```

або для передачі стандартного виводу команди на стандартний вхід списку команд

```
>(список команд)
```

Зауваження, між символами >, < і (не має бути пропусків.

Процес заміни використовує файли `/dev/fd/<n>` для направлення результатів команд в дужках на вхід іншої команди.

```
$ echo >(true)
/dev/fd/63

$ echo <(true)
/dev/fd/63

$ echo >(true) <(true)
/dev/fd/63 /dev/fd/62

$ wc <(cat /usr/share/dict/linux.words)
 483523  483523 4992010 /dev/fd/63

$ grep script /usr/share/dict/linux.words | wc
    262     262    3601

$ wc <(grep script /usr/share/dict/linux.words)
    262     262    3601 /dev/fd/63
```

Заміна процесу може порівнювати вивід двох різних команд або навіть вивід різних параметрів для однієї команди. Так команда `comm` читає два файли або потоки, які мають бути попередньо лексично відсортовані, і генерує вивід, що складається з трьох стовпців тексту:

- рядки, знайдені тільки у файлі *файл1*;
- рядки, знайдені тільки у файлі *файл2*;
- рядки, спільні для обох файлів.

```
$ comm <(ls -l) <(ls -al)
```

Наприклад, можна передати вивід декількох команд `find` на вхід команді `wc`:

```
$ wc -l \
    <(find / -mindepth 1 -maxdepth 1 -type d) \
    <(find /opt -mindepth 1 -maxdepth 1 -type d)
     20 /proc/self/fd/11
      2 /proc/self/fd/12
     22 total
```

У цьому прикладі використовується команду `find` для отримання кількість каталогів у каталогах `/` і `/opt` .

12. Співпрограми

Коли два (або більше) процеси явно запрограмовані на одночасний запуск і, можливо, взаємодіють один з одним, їх називають співпрограмами (англ. coroutines).

Якщо процеси не взаємодіють із собою, то навіщо потрібно програмувати співпрограми, які не спілкуються одна з одною? Кожний процес можна характеризувати з точки зору використання системних ресурсів:

- інтенсивно використовує процесор;
- має багато введенням/виведення даних;
- інтерактивний (вимагає втручання користувача).

Використання співпрограм, які не взаємодіють між собою дозволяє ефективно використовувати системні ресурси.

Розглянемо дві команди, які взаємодіють між собою (результат виконання першої команди використовує друга команда. Команди виконуються в одній сесії оболонки, але перша – на задньому плані, а друга на передньому плані.

```
#!/bin/bash
cat <(sleep 5; echo "Привіт світ") > out &
nl out
```

Сценарій завершиться швидше ніж перша команда, тому друга команда нічого не виведе. Для того, щоб сценарій почекав завершення першої команди потрібно добавити команду `wait`.

```
#!/bin/bash
cat <(sleep 5; echo "Привіт світ") > out &
wait
nl out
```

```
1 Привіт світ
```

Більші можливості для роботи оболонки із спіпроцесами надає команда `coproc`. Синтаксис команди

```
coproc command args         #перший синтаксис
coproc <name> command args   #другий синтаксис
```

В першому варіанті під час виконання співпроцесу створюється масив із назвою COPROC за замовчуванням, у другому варіанті назва <name> задається. Перший елемент цього масиву є вихідним дескриптором, тоді як другий елемент масиву є вхідним дескриптором для спільного процесу. Двонаправлений канал встановлюється між виконавчою оболонкою та співпроцесом. Bash поміщає дескриптори файлів для цих каналів у масив:

<name>[0] – це дескриптор файлу для каналу, який підключено до стандартного виводу співпроцесу у виконавчій оболонці.

<name>[1] – це дескриптор файлу для каналу, підключеного до стандартного входу співпроцесу у виконавчій оболонці.

```
coproc (echo $whoami))
echo "COPROC масив ${COPROC[*]}"
echo "COPROC PID ${COPROC_PID}"
read -r -p "Введіть ім'я користувача: " o >& "${COPROC[0]}"
echo "Вихід співпроцесу o=$o"
```

```
COPROC масив 63 60
COPROC PID 16426
Введіть ім'я користувача: xxx
Вихід співпроцесу o=xxx
```

Приклад надсилання і отримання змінної із співпроцесу з іменем `my_coproc`, заданим користувачем

```
coproc my_coproc { bash ; }
var=1.234
echo "echo ${var}" >& "${my_coproc[1]}"
read var <& "${my_coproc[0]}"
echo $var
```

```
1.234
```

Приклад виконання співпрограми `tr`, яка замінює у вхідній стрічці символ "a" на символ "b".

```
coproc my_coproc { tr a b; }
echo "aa-bb-aa-bb-ab" >& "${my_coproc[1]}"
exec {my_coproc[1]}>&-
cat <& "${my_coproc[0]}"
```

```
bb-bb-bb-bb-bb
```

Для спрощення взаємодії батьківської оболонки і процесу, запущеного на задньому плані за допомогою додавання до команди `!&` замість `&`, використовуються команди `read -p` і `print -p`, рис. 1.

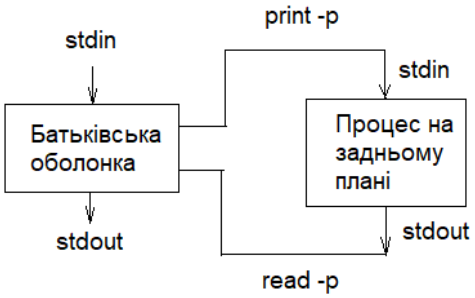

Рисунок 1 – Взаємодія батьківської оболонки і процесу на задньому плані

Запитання.

1. Вказати пристрої файли і дати їх характеристику.

2. Які імена, номери та призначення стандартних файлових дескрипторів.
3. Перенаправлення потоків з використанням символів >, >>, <, <<, |.
4. Створення реєстраційних файлів.
5. Використання команди exec
6. Тимчасове перенаправлення файлових дескрипторів.
7. Постійне перенаправлення файлових дескрипторів.
8. Створення власних перенаправлень для потоків введення і виведення.
9. Створення дескрипторів з можливістю читання-записування у файли.
10. Отримання інформації про відкриті файлові дескриптори процесів Linux.
11. Створення і використання тимчасових каталогів і файлів.
12. Іменовані канали.
13. Тимчасові іменовані канали.
14. Анонімні канали.
15. Співпрограми.

8. 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.

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, 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 командна оболонка (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.

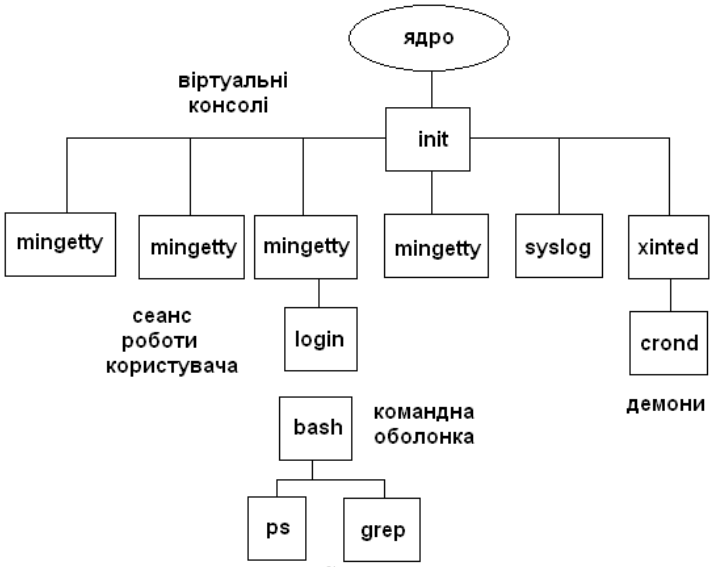

Рисунок 1 – Фрагмент ієрархії процесів

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 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 командна оболонка користувача (shell), яка вказана 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.

Запуск init-сценаріїв.

Усі 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 оболонці 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 переривається 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 переривається 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|функція 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, поточна командна оболонка 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 дистрибутиву 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 дистрибутиву 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.

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.

9. ФУНКЦІЇ 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 командного рядка у сценарій і функцію
3. Глобальні і локальні змінні
4. Передача масивів і функції
5. Повернення значень із функції
6. Рекурсія
7. Команда source
8. 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
# використання функції у сценарії
function my_fun {
```

```
  echo "count=$count"
}
count=0
while [ $count -le 3 ]
do
  my_fun
  count=$[ $count + 1 ]
done
echo "Кінець циклу"
my_fun
echo "Кінець сценарію"
```

```
$ ./test1.bash
count=0
count=1
count=2
count=3
Кінець циклу
count=4
Кінець сценарію
```

За замовчуванням код завершення функції (exit status) є кодом, який повертає остання команда функції. Після завершення функції можна використати зарезервовану змінну **$?** для отримання коду повернення функції.

```
#!/bin/bash
# test2.bash - тестування exit статусу функції
my_func() {
  echo "виведення інформації про неіснуючий файл"
  ls -l badfile
}
echo "тестування функції:"
my_fun
echo "exit статус: $?"
```

```
$ ./test2.bash
тестування функції:
виведення інформації про неіснуючий файл
ls: cannot access 'badfile': No such file or directory
exit статус: 2
```

Команда `return` дозволяє повернути із функції цілочисловий код із значенням від 0 до 255:

```
#!/bin/bash
# test3.bash - використання return команди у функції
function my_fun {
  read -p "Введіть значення: " value
  echo "подвоєння значення"
  return $[ $value * 2 ]
}
my_fun
echo "Нове значення $?"
```

```
./test3.bash
Введіть значення: 3
подвоєння значення
```

```
6
```

Результат виконання функції `my_fun`, який повертає команда `echo`, можна зберегти у змінній:

```
#!/bin/bash
# test4.bash - використання echo для повернення значення
function my_fun {
  read -p "Введіть значення: " value
  echo $[ $value * 2 ]
}
result=$(my_fun)
echo "Нове значення $result"
```

```
$ ./test4.bash
Введіть значення: 2
Нове значення 4
```

Функцію можна експортувати з поточної оболонки у підоболонку за допомогою команди

```
export -[f] -[n] -[p] [name=[value] ...]
```

`-f` експорт [name] як ім'я функції;
`-n` вилучення [name] із списку експортованих змінних;
`-p` виведення списку експортованих функцій і змінних у поточному сеансі оболонки.

```
export -f my_fun
```

Команда `source` (синонім `.`) вставляє функцію із файлу у сценарій

```
#!/bin/bash
# test5.bash - виклик функції my_fun з файлу
. my_fun
result=$(my_fun)
```

```
$ ./test5.bash
Введіть значення: 2
Нове значення 4
```

2. Передача параметрів командного рядка у сценарій і функцію

Якщо сценарій запустити з параметрами командного рядка, то вони присвоюються змінним середовища Bash `$0, $1,...`,

```
#!/bin/bash
echo "$1 $2 $3"
```

```
1 2 3
```

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 у функцію параметрів командного рядка:

```
#!/bin/bash
# test6.bash - передача у функцію параметрів командного рядка
function addem {
  if [ $# -eq 0 ] || [ $# -gt 2 ]
```

```
  then
    echo -1
  elif [ $# -eq 1 ]
  then
    echo $(( $1 + $1 ))
  else
    echo $(( $1 + $2 ))
  fi
}
declare -x -f addem
echo -n "Додати 10 і 15: "
value=$(addem 10 15)
echo $value
25

...
echo -n "Додати одне число: "
$ value=$(addem 10)
$ echo $value
20

...
$ echo -n "Додати 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
# test7.bash - доступ до параметрів сценарію всередині функції
function my_fun {
  echo $(( $1 * $2 ))
}
# $# - кількість аргументів
if [ $# -eq 2 ]
then
  value=$(my_fun $1 $2)
  echo "Результат $value"
else
  echo "Запуск: test7.bash <i> <i>"
fi

$ ./test.bash
Запуск: test7.bash <i> <i>

$ ./test7.bash 2 3
Результат 6
```

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 змінних визначених за межами функції можна звертатися із функції:

```
#!/bin/bash
# test8.sh використання глобальної змінної для передачі значення
function my_fun {
  value=$(( $value * 2 ))
}
read -p "Введіть значення: " value  # глобальна змінна сценарію
my_fun
echo "Нове значення: $value"
```

```
$ ./test.sh
Введіть значення: 4
Нове значення: 8
```

Люба змінна, яка використовується тільки всередині функції оголошується локальною:

```
local temp
```

Ключове слово `local` може використовуватися при присвоєнні значення змінній:

```
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 змінні. Так в наступному сценарії використовуються дві різні змінні з однаковим іменем `temp`:

```
#!/bin/bash
# test9.sh - приклад локальної змінної
function my_fun {
  local temp=5  # temp локальне
  echo "temp=$temp локальне"
  temp_loc=$temp
}
temp=10    # temp глобальне
echo "temp=$temp глобальне"
my_fun
if [ $temp -gt $temp_loc ]
```

```
then
  echo "temp=$temp є більшим temp_loc=$temp_loc"
else
  echo "temp=$temp є меншим  temp_loc=$temp_loc"
fi
```

```
$ bash test9.sh
temp=10 глобальне
temp=5 локальне
temp=10 є більшим temp_loc=5
```

Кожна функція має атрибути. Команда `declare (typeset)` призначає функціям атрибути. Синтаксис команди

```
declare [опції] [ім'я-змінної] "[значення]"
```

Атрибути (опції) для функцій:

`-f` оголошення функції

`-F` виводить імена функції і атрибути

`-g` застосовує глобальну область видимості для всіх змінних всередині функції

4. Передача масивів у функції

Якщо спробувати передати як параметр у функцію масив, то функція прийме тільки перший елемент масиву:

```
#!/bin/bash
# test10.sh - передача масиву у функцію
function my_fun {
echo "Параметри отримані у функції: $@"
thisarray=$1
echo "Отриманий масив ${thisarray[*]}"
}
myarray=(1 2 3 4 5)
echo "Оригінальний масив: ${myarray[*]}"
my_fun $myarray
```

```
$ ./test10.sh
Оригінальний масив: 1 2 3 4 5
Параметри отримані у функції: 1
Отриманий масив 1
```

Тому, для передачі масиву у функцію потрібно видобувати значення елементів масиву в окремі значення і передавати їх як параметр 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.

Змінна **`$*`** містить усі параметри командного рядка як одну стрічку.

Змінна **`$@`** теж містить усі параметри командного рядка, але кожний параметр як окрему стрічку.

Розширення змінних **`$*, "$*", $@, "$@"`** показує наступний приклад

```
# test.sh
#!/bin/bash
echo "Using \"\$*\":"
```

```
for a in "$*"; do
    echo $a;
done

echo -e "\nUsing \$*:"
for a in $*; do
    echo $a;
done

echo -e "\nUsing \"\$@\":"
for a in "$@"; do
    echo $a;
done

echo -e "\nUsing \$@:"
for a in $@; do
    echo $a;
done
```

```
$ bash test.sh один два "три чотири"
# параметри розглядаються як одна стрічка у лапках
Using "$*":
один два три чотири

# стрічка розбити на слово for циклом
Using $*:
один
два
три
чотири

# кожний елемент, як окрема стрічка
Using "$@":
один
два
три чотири

# кожний елемент, як окрема стрічка без лапок, тому останній елемент розбитий на дві стрічки
Using $@:
один
два
три
чотири
```

Функція відновлює змінну масив з отриманих параметрів командою `echo $@` і використовує його для обчислення суми елементів:

```
#!/bin/bash
# test11.sh - обчислення суми елементів масиву
function my_fun {
  local sum=0
  local newarray
  newarray=($(echo "$@"))  # усі параметри консольного рядка як масив
  for value in ${newarray[*]}  # всі елементи масиву
  do
    sum=$[ $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"
```

```
$ ./test11.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
# test12.sh - returning an array value
function my_fun {
  local origarray
  local newarray
  local elements
  local i
  origarray=($(echo "$@")) # локальний оригінальний масив у функції
  elements=$(( $# - 1 ))   # кількість елементів масиву -1, індексація з 0
  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 ${myarray[*]})  # всі елементи масиву як одна стрічка
result=($(my_fun $arg1))  # створення нового масиву у сценарії
echo "Масив із функції: ${result[*]}"
```

```
$ ./test12.sh
Оригінальний масив: 1 2 3 4 5
Масив із функції: 2 4 6 8 10
```

Сценарій передає значення масиву `myarray` як одну стрічку, використовуючи змінну `arg1`, у функцію `my_fun`. Функція відновлює масив у новий масив `origarray`. Функція використовує команду `echo "$@"` для виведення значень у масив `newarray`. Сценарій використовує результат виконання функції `my_fun` для створення нового масиву `result`.

5. Повернення значень із функції

Повернення одного значення із функції.

Функції Bash, на відміну від функцій у більшості мов програмування, не дозволяють повертати значення у викликаючу функцію. Коли функція Bash завершує роботу, вона повертає значення статусу: нуль для успіху, не нуль для невдачі. Щоб повернути значення, можна

встановити глобальну змінну з результатом, використати підставлення команди або передати ім'я змінної для використання змінної, як результату.

Bash оператор `return` повертає статус завершення функції, який є числовим значенням від 0 до 255, вказаного в операторі виходу. Значення статусу зберігається у змінній середовища `$?`. Якщо функція не містить оператора `return`, її статус встановлюється на основі статусу останнього виконаного у функції оператора. Щоб фактично повернути довільні значення потрібно використовувати інші механізми.

Найпростіший спосіб повернути значення з функції – просто встановити глобальну змінну для результату. Оскільки всі змінні в bash є глобальними за замовчуванням, це зробити легко:

```
out=""
function myfunc()
{
    local name=$1  // параметр функції
    out="Привіт ${name}"
}
# виклик функції з аргументом
myfunc Іван
echo "${out}"
```

Глобальна змінна `out` використовується як результат виконання функції. Однак, використання глобальних змінних, особливо у великих програмах, може призвести до виникнення помилок, які важко знайти.

Кращим підходом є використання локальних змінних у функціях. Для повернення результату використовується підставлення команд:

```
function myfunc()
{
    local name=$1  # параметр функції
    echo "Привіт ${name}"
}
# виклик функції з аргументом
echo $(myfunc Іван)
```

Тут використовується підставлення команди `myfunc`, щоб отримати значення, яке повертає функція.

Інший спосіб повернути значення – написати функцію так, щоб вона приймала назву змінної як частину командного рядка, а потім встановити для цієї змінної результат функції:

```
function myfunc()
{
    local  __resultvar=$1
    local  myresult='some value'
    eval $__resultvar="'$myresult'"
}

myfunc result
echo $result
```

Оскільки ім'я змінної, яку потрібно встановити, зберігається у змінній командного рядка, її не можна встановити безпосередньо, тому потрібно використовувати `eval`. Оператор eval в вказує Bash інтерпретувати рядок двічі. Перша інтерпретація призводить до рядка `result='some value'`, який потім інтерпретується ще раз і завершується встановленням змінної абонента.

Повернення декількох значень із функції.

```
output=$(my_func)
echo $output
result1=$(echo $output | awk '{print $1}')
result2=$(echo $output | awk '{print $2}')
echo "Result 1: $result1"
echo "Result 2: $result2"
```

```
value1 value2
Result 1: value1
Result 2: value2
```

Повернення глобального масиву із функції.

```
declare -a array
function get_array(){
  array=("apple" "banana" "orange" "cherry")
}

get_array
echo "${array[0]}"
echo "${array[1]}"
echo "${array[2]}"
echo "${array[3]}"
```

apple
banana
orange
cherry

Повернення локального масиву із функції.

```
#!/bin/bash

function return_array(){
 local array=(1 2 3 4 5)
 for element in "${array[@]}"; do
  echo "$element"
 done
}

# Виклик функції і захоплення результату використовуючи підставлення команд
result=$(return_array)
echo $result
```

1 2 3 4 5

Повернення локального масиву із функції з використання підставлення процесів.

```
my_function() {
  local -a result=("value1" "value2" "value3")
  echo "${result[@]}"
}

# Захоплення виходу функції
readarray -t results < <(my_function)
echo "${results[@]}"
```

value1 value2 value3

Повернення локального масиву із функції з використання підставлення процесів.

```
#!/bin/bash

function get_array() {
 local array=("apple" "banana" "orange" "cherry")
 for element in "${array[@]}"; do
    echo "$element"
 done
}

result=$(< <(get_array))
echo "$result"
```

```
apple
banana
orange
cherry
```

*Повернення локального масиву із функції з використання **IFS** змінної.*

У сценарії спочатку визначається функція під назвою `return_array`, яка створює локальну змінну масиву під назвою `` `array` ``, що містить цілі числа від 1 до 3. `echo "${array[@]}"` виводить всі елементи масиву, розділені пропусками. Потім захоплюється вихід функції `return_array` у змінну `` `array` `` за допомогою підставлення команд. Після цього `ind_elems=($array)` розбиває `` `array` `` на окремі елементи за допомогою значення `IFS`. Нарешті, виконується ітерація і друк кожного елементу масиву "ind_elems" за допомогою циклу `for`.

```
#!/bin/bash

function return_array(){
 local array=(1 2 3)
 echo "${array[@]}"
}

IFS=' '
array=$(return_array)
ind_elems=($array)
 for elem in "${ind_elems[@]}"; do
  echo "$element"
 done
```

```
1
2
3
```

*Повернення локального масиву із функції з використання посилання **nameref** на масив.*

У функції `get_array local -n array=$1` оголошується локальна змінна "`array`", а параметр `-n` робить її посиланням `nameref`. Потім цьому `nameref` присвоюється значення першого аргументу (`$1`), наданого функції. Це означає, що "`array`", який містить 4 елементи `("apple" "banana" "orange" "cherry")`, стає посиланням на масив, переданий у функцію.

Після цього `declare -a result_array` оголошує "`result_array`", який містить результат, повернутий функцією "`get_array`". Отже, `get_array result_array` викликає функцію `get_array` і передає "`result_array`" як аргумент, дозволяючи "`get_array`" змінювати

“result_array” безпосередньо через nameref. Нарешті, цикл for виконує ітерацію по елементах масиву та друкує їх у терміналі.

```
#!/bin/bash
# визначення функції
function get_array {
 local -n array=$1  # створення посилання nameref
 # використання nameref для призначення значень
 array=("apple" "banana" "orange" "cherry")

}

# оголошення масиву для результату
declare -a result_array

# виклик функції з посиланням на result_array
get_array result_array
for element in "${result_array[@]}"; do
 echo "$element"
done
```

```
apple
banana
orange
cherry
```

Повернення масиву із функції створеного here-документом (англ. here-document.

У цьому сценарії функція return_array використовує команду cat для відображення вмісту в межах розділювача EOF…EOF. Потім mapfile -t my_array < <(return_array) захоплює вихідні дані функції та перетворює текст на масив. Параметр -t вилучає кінцеві символи нового рядка з кожного прочитаного рядка. echo "Array elements: ${my_array[@]}" друкує всі елементи масиву “my_array”.

```
#!/bin/bash

# Функція використовує a here-документ для повернення масиву
function return_array {
 cat <<EOF
1
2
3
4
5
EOF
}

# Захоплення виходу у масив за допомогою mapfile
mapfile -t my_array < <(return_array)

# Друк елементі масиву
echo "Array elements: ${my_array[@]}"
```

```
Array elements: 1 2 3 4 5
```

*Повернення масиву із функції з використанням команди **printf**.*

Команда `printf` у Bash виводить текст або змінні на стандартний вивід. Поєднання команди `printf` із функцією забезпечує повернення масиву з функції. Функція `get_array` містить локальну змінну "`array`", яка складається з 4 елементів. Після виклику функції `printf "%s\n" "${array[@]}"` друкує всі елементи масиву в окремих рядках в терміналі.

```
#!/bin/bash

get_array() {
 local array=("apple" "banana" "orange" "cherry")
 printf "%s\n" "${array[@]}"
}

get_array
```

```
apple
banana
orange
cherry
```

6. Рекурсія

Локальні змінні функцій мають властивість самовмісткості. Самовмісткі 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`:

```
#!/bin/bash
# test13.sh - використання рекурсії
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"
```

```
$ ./test13.sh
Введіть значення: 5
Факторіал 5: 120
```

7. Команда source

Команда `source` є вбудованою командою Bash і призначена для завантаження функцій, змінних і конфігураційних файлів у сценарій.

Синтаксис команди

```
source filename [arguments]
```

```
. filename [arguments]
```

Команда `source` має також синонім «.», як команда крапка.

Якщо `filename` не є повним шляхом до файлу, команда шукатиме файл у каталогах, указаних у змінній середовища `$PATH`. Якщо файл не знайдено в `$PATH`, команда шукатиме файл у поточному каталозі.

Якщо задано будь-які `arguments`, вони стануть позиційними параметрами для `filename`.

Якщо `filename` існує, команда source повертає 0, інакше, якщо файл не знайдено, команда повертає 1.

Приклад використання функції `check_root()`, записаної у файл `functions.sh`.

```
# functions.sh
check_root () {
  if [[ $EUID -ne 0 ]]; then
    echo "This script must be run as root"
    exit 1
  fi
}
```

Тепер кожний сценарій, який може запускати тільки `root` користувач, повинен містити наступні рядки

```
#!/usr/bin/env bash
source functions.sh
check_root
echo "Сценарій виконує root"
```

За допомогою `source` команди можна створювати конфігураційні файли сценаріїв

Приклад конфігураційного файлу сценарію `config.sh`

```
user="Іван"
psw="1234"
```

Використання конфігураційного файлу

```
#!/usr/bin/env bash
source config.sh
echo "user=$user"
echo "psw=$psw"
```

```
Іван
1234
```

Для виклику функцій у різних сценаріях можна створити окремий файл, наприклад з іменем `myfuncs`, який містить потрібні функції

```
# 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
}
```

Команда `source` завантажить функції у потрібний сценарій

```
#!/usr/bin/env bash
# test14.sh - використання функцій з бібліотеки myfuncs
source ./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"
```

```
$ ./test14.sh
Результат додавання: 15
Результат множення: 50
Результат ділення: 2
```

Недолік такого підходу у втраті визначення функцій при перевантаженні Bash оболонки. Це може бути суттєвим недоліком для складних сценаріїв. У таких випадках шлях до бібліотечного файлу потрібно вказати 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 і при перевантаженні Bash оболонки.

8. 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`.

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. ІНТЕРАКТИВНІ КОМАНДИ. КЕРУВАННЯ КОЛЬОРОМ

Мета. Вивчення засобів інтерактивної взаємодії з використанням текстових списків вибору, керування режимами відображення кольорів дисплею, застосування команди `dialog` для створення діалогових вікон.

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\tЗробити вибір: "
```

Остання команда `echo` має опцію `-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 introducer), яка вказує, що за нею розміщуються параметри, які встановлюють режим відображення дисплею та задають колір тексту і фону.

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 консолі або таких редакторах використовується послідовність натискання клавіш `Ctrl-v`, а потім `Esc`. В результаті появиться символ **`^[`** до якого потрібно додати **`[`**, що дає **`^[[`**.

У командах `echo -e` і `printf` Esc-символи **`^[[`** можна задати послідовністю символів **`` `\e[` ``** або **`` `\033[` ``**.

Після керуючої послідовності CSI задаються параметри SGR (Select Graphic Rendition), які керують відображенням дисплею. Синтаксис параметрів SGR:

```
CSI n[;k]m
```

`n` - параметри, які визначають коди кольорів;

`k` - параметри, які визначають режими відображення;

`m` – признак кінця SGR параметрів.

Значення кодів кольорів показані в табл. 1.

Таблиця 1 – Коди ANSI кольорів

Код	Описання
0	Чорний
1	Червоний
2	Зелений
3	Жовтий
4	Голубий
5	Бордовий
6	Бірюзовий
7	Білий
8	Зміна кольору

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:

CSI31m – червоний текст
CSI47m – білий фон

Можна об'єднати колір тексту і фону:
CSI31;47m – текст червоний, фон – білий.

```
$echo -e '\e[31m Це червоний текст на чорному фоні \e[0m'
$echo -e '\e[31;47m Це червоний текст на білому фоні \e[0m'
```

Зміна RGB кольору:

Встановити 8-бітовий RGB колір тексту (код 5) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[38;5;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Встановити 8-бітовий RGB колір фону (код 5) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[48;5;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Встановити 24-бітовий RGB колір тексту (код 2) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[38;2;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Встановити 24-бітовий RGB колір фону (код 2) Red=15; Green=194; Blue=40:

```
echo -e "\n\e[48;2;35;194;40m RGB 8-біт \e[0m встановлено! \n"
```

Значення кодів режимів відображення показані в табл. 2.

Таблиця 2 – Коди режимів 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

```
echo -e '\n\e[31;1m Червоний жирний \e[0m \n'
echo -e "\n\e[31;4m Червоний підкреслений \e[0m \n"
echo -e "\n\e[31;47;5m Блимаючий червоний на білому фоні \e[0m \n"
```

Для зміни кольорів тексту і фону можна використати команду `tput`:
`tput setaf color` – задати колір тексту;
`tput setab 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.

```
tput setaf 1
tput setab 3
echo "Друк червоного тексту на жовтому фоні"
tput sgr0
```

Зміна кольору повідомлення командного рядка:

```
'\e[x;ym; $PS1 \e[0m'
```

де `x,y` - режими відображення.

Виведення PS1:

```
user@HP:~$ echo $PS1
```

```
\[\e]0;\u@\h:  \w\a\]${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\
033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$
```

Зміна кольору повідомлення з використанням команди `export`:

```
$ export PS1='\e[1;31m[\d \t \u@\h \w]\$ \e\0m'
```

```
$ export PS1=’\e[1;31m\u@h:\w\e[0m\$’
```

де `\e[1;31m` - встановлення яскравого червоного кольору тексту;

`\d` – дата;

`\t` - час;

`\u` – користувач;

`\h` – хост;

`\w` – повний шлях (`\W` – остання частина повного шляху до `/home/user`).

визначення режимів курсора:

```
CSI n q
```

n - код режиму курсора, табл. 3.

Таблиця 3 – Коди режимів курсора

Код.	Описання
1	Блимаючий блок
2	Нерухомий блок (за замовчуванням)
3	Блимаюче нижнє підкреслення
4	Нерухоме нижнє підкреслення
5	Блимаючий вертикальний прямокутник
6	Нерухомий вертикальний прямокутник

```
$ echo -e '\e[4 q'
```

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 "\e[1m\t0. Exit program\n\n\e[0m\e[44;33m"
  echo -en "\t\tEnter option: "
  read -n 1 option
}

echo "\e[44;33m"
while [ 1 ]
  do
    menu
    case $option in
    0)
       break ;;
    1)
       diskspace ;;
    2)
       whoseon ;;
    3)
       memusage ;;
    *)
    clear
    echo -e "\e[5m\t\t\tSorry, wrong selection\e[0m\e[44;33m";;
    esac
    echo -en "\n\n\t\t\tHit any key to continue"
    read -n 1 line
done
echo "\e[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 – Опції 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	Висвітити 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 тегу 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 діалогового 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 діалогу 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.

Запитання.

1. За допомогою яких команд Bash можна створювати текстові меню.

2. Які переваги команди `select` при створенні текстових меню.

3. Як поміняти кольори і режими роботи дисплею з використанням керуючих ANSI `Esc`-символів за допомогою команд `echo` і `printf`.

4. Як поміняти кольори тексту за допомогою команди `tput`.

5. Як поміняти кольори повідомлення командного рядка з використанням керуючих ANSI `Esc`-символів.

6. Як поміняти режим відображення курсора командного рядка з використанням керуючих ANSI `Esc`-символів.

7. Як створити віконні діалоги за допомогою команди `dialog`.

11. ПОТОКОВІ РЕДАКТОРИ SED І 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

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,md (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 вставляється 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 файлу 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 qawk '{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 степені 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 співпадає 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 індексу і значення. Індексом може бути як число, так і стрічка.

```
Index 3 Value 30          Index "dog" Value "chien"
Index 1 Value "foo"       Index "cat" Value "chat"
Index 0 Value 8           Index "one" Value "un"
Index 2 Value ""          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) }

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`.

12. МЕРЕЖЕВІ КОМАНДИ

Мета. Вивчення можливостей Bash і команд ОС Linux для роботи в мережі

Вступ. Оболонка Bash має обмежені вбудовані можливості для роботи з TCP сокетами. Команди ОС Linux мають значно більше можливостей для роботи з мережею, сокетами TCP/UDP, серверами. Використання команд ОС Linux в сценаріях Bash дозволяють розробляти повноцінні мережеві засоби.

План.

1. Команди для роботи в мережі
2. Стратегії клієнт-сервер і рівний-до-рівного (лорд-лорд, peer-to-peer)
3. IP-адреса і сокети
4. Синтаксис мережевих сокетів
5. Bash клієнт. Читання і записування у відкритий сонет
6. Сервер і клієнт з використанням команди `netcat`
7. Команда `netstat`
8. Команда ss
9. Команда налаштування мережі ip
10. CGI-сценарій
    10.1. Змінні середовища CGI-програми
    10.2. Оброблення форм

1. Команди для роботи в мережі

Для отримання даних і роботи з мережею використовуються команди `ip`, `netstat`, `ss`, `netcat`, `iptables`, `lsof`, `curl`, `wget`.

`ip` - робота з маршрутами, пристроями,тунелями.

`netstat` – відображення поточного статусу підключень по TCP/IP або UDP, таблиць маршрутизації, кількості адаптерів та статистики протоколів.

`ss` - виведення статистики сокетів (заміна для команди `netstat`).

`netcat` - робота з TCP/UDP з'єднаннями,.HTTP серверами.

`iptables` - підтримка мережевих сокетів.

`lsof` – підтримка списку всіх відкритих файлів і процесів, які їх відкрили.

`curl` – передача та отримання даних з сервера з використанням протоколів DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, LDAP, LDAPS, POP3, RTMP, SCP, SFTP, SMTP, TELNET, TFTP.

`wget` – завантаження файлів з web серверів.

2. Стратегії клієнт-сервер і рівний-до-рівного (лорд-лорд, peer-to-peer)

У стратегії клієнт-сервер одна програма назначається сервером, а інша клієнтом. Сервер прослуховує порти на запити, обробляє їх і відправляє відповіді. Клієнти надсилають запити і отримують відповіді. Веб-сервер і веб-переглядач є прикладом такої пари. Переглядач завжди

ініціює обмін даними з веб-сервером. Стратегія клієнт-сервер є найбільш поширеною через її простоту.

Засобами Bash можна написати тільки клієнт, так як він не має можливостей для прослуховування вхідних з'єднань для заданого порту. Але у Bash сценарії можуть використовуватися команди Linux, які мають такі можливості.

У стратегії рівний-до-рівного (peer-to-peer) в одній програмі інтегровано клієнт і сервер, шляхом призначення функцій клієнта і сервера різним портам. Така програма може приймати і відправляти запити. Ця стратегія використовується для розподілення великих задач або даних між декількома комп'ютерами, працюючими паралельно.

3. IP-адреса і сокети

Кожний комп'ютер в мережі має унікальну IP-адресу і одне або декілька імен хоста `hostname`. `IP`-адресу за іменем хосту `hostname` можна отримати командою `host`

```
$ host pnu.edu.ua
pnu.du.ua has address 194.44.152.137
```

Локальну адресу хосту можна отримати командою

```
> ip addr
etho: …
inet 192.168.1.101
```

Коли комп'ютер працює не в мережі використовується віртуальний інтерфейс `loopback`, призначений для внутрішнього зв'язку. Інтерфейс запускається на хості з іменем `localhost` і IP-адресою 127.0.0.1. Реальна фізична адреса інтерфейсу (`eth0`) 192.168.1.1.

Для мережевого зв'язку між комп'ютерами використовують протокол Інтернету і локальних мереж (LAN) TCP/IP (Transport Control Protocol/Internet Protocol). В цього протоколі важливим поняттям є сокет (socket).

Сокет – це мережева структура даних, що реалізує поняття "Кінцевої точки зв'язку". Перш ніж встановити зв'язок, мережеві застосування спочатку повинні створити сокети. Їх можна порівняти з телефонними розетками, без підключення до яких неможливо увійти до телефонної мережі. Сокети бувають двох різновидів: файлові і мережеві.

Можна відкрити мережеві TCP/IP сокети у Bash використовуючи спеціальний шлях `/dev/tcp`. Це не шлях до реального файлу пристрою, а Bash інтерпретує його як запит на відкриття сокету. Bash може також створити UDP (User Datagram Protocol) сокет використовуючи шлях `/dev/udp`. Однак, UDP використовуються для надсилання коротких повідомлень, частина яких може не прибувати до місця призначення.

На додаток до протоколу і соку потрібний порт. Порт є числом, яке використовуються для ідентифікації різних служб на віддаленому комп'ютері. Наприклад, веб-сервер звичайно використовує порт з номером 80. Для Bash сценарію сокет є реальним файлом з незвичайним шляхом. Сокет діє як іменований канал, заблокований, поки в ньому є нова інформація для зчитування. Коли сокет закритий, команда `read` повертає признак досягнення кінця файлу. На відміну від каналів сокет завжди доступний для читання і запису.

4. Синтаксис мережевих сокетів

Для відкриття мережевих сокетів TCP або UDP використовується наступний синтаксис:

```
exec {file-descriptor}<>/dev/{protocol}/{host}/{port}
```

`file-descriptor` – файловий дескриптор який асоціюється із сокетом і має починатися з 3, так як дескриптори 0, 1, 2 зарезервовані за `stdin, stdout i stderr`.

`<,>,<>` - дескриптор відкритий для читання, записування, читання/записування;

`protocol – tcp` або `udp`.

Після завершення передачі даних відкритий сокет має бути закритий з використанням наступного синтаксису:

```
$ exec {file-descriptor}<&-
$ exec {file-descriptor}>&-
```

Перша команда закриває вхідне з'єднання, а друга – вихідне.

Створення сокету TCP для читання і записування:

```
exec 3<>/dev/tcp/pnu.edu.ua/80
```

5. Bash клієнт. Читання і записування у відкритий сокет

Запис повідомлення із змінної $MESSSAGE у сокет:

```
> echo -ne $MESSAGE >&3
> printf $MESSAGE >&3
```

Читання повідомлення із сокету у змінну:

```
> read -r -u -n $MESSAGE <&3
> MESSAGE=$(dd bs=$NUM_BYTES count=$COUNT <&3 2> /dev/null)
```

Приклади.

1. Завантаження віддаленої web-сторінки і її друк:

```
# 1.sh
# 2-й рядок відкрити файловий дескриптор 3 для читання/записування у сокет TCP/IP
# 3-й рядок відправити запит у сокет
# 3,4-й прочитати відповідь і вивести на екран
#!/bin/bash
exec 3<>/dev/tcp/pnu.edu.ua/80
echo -e "GET /HTTP/1.1\r\nhttps://pnu.edu.ua\r\nConnection: close\r\n\r\n">&3
msg="$(cat <&3)"
echo $msg
./1.sh
HTTP/1.1 400 Bad Request
Date: Tue, 27 Nov 2018 11:00:19 GMT
Server: Apache
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

2. Висвітлення версії віддаленого SSH:

```
# 2.sh
#!/bin/bash
exec 3< /dev/tcp/192.168.0.10/22
timeout 1 cat <&3
```

В дійсності цей сценарій може бути скорочений до

```
#!/bin/bash
timeout 1 cat </dev/tcp/192.168.0.10/22
```

3. Друк поточного часу з сервера точного часу:

```
# 3.sh
#!/bin/bash
cat </dev/tcp/time.nist.gov/13

./3.sh
58449 18-11-27 10:34:41 00 0 0 925.7 UTC(NIST) *
```

4. Перевірка Інтернет з'єднання:

```
# 4.sh
#!/bin/bash
HOST=pnu.edu.ua
PORT=80
(echo >/dev/tcp/${HOST}/${PORT}) &>/dev/null
if [ $? -eq 0 ]; then
   echo "Connection successful"
else
   echo "Connection unsuccessful"
fi
./4.sh
Connection successful
```

5. Сканування портів віддаленого хоста:

```
# 5.sh
#!/bin/bash
host=$1
port_first=1
port_last=2000
for ((port=$port_first; port<=$port_last; port++))
do
  (echo >/dev/tcp/$host/$port) >/dev/null 2>&1 && echo "$port open"
done
> ./5.sh localhost
25 open
631 open
1716 open
```


6. Сервер і клієнт з використанням команди `netcat`

Команда `netcat (nc)` використовується для роботи з TCP і UDP сокетами, HTTP серверами. При роботі із сокетами використовуються наступні опції:

-4 – адресація IP4;

-6 – адресація IP6;

-i інтервал – затримка між відправленням і отриманням пакету;

-k – заставляє команду продовжувати прослуховувати інші зєднання, після завершення поточного;

-l – прослуховування вхідних з'єднань;

-n – не використовувати DNS та інші сервіси;
-p – номер використовуваного порту;
-s – IP-адреса для відправлення пакетів;
-u – використання UDP пакетів замість TCP;
-w timeout – час закриття з'єднання, яке знаходиться в режимі очікування.

Приклад. Простий сервер і клієнт TCP для передачі повідомлення:
Консоль 1. Сервер:
```
> nc -lk localhost 3000
```
Консоль 2. Клієнт:
```
> nc localhost 3000
Привіт
світ!
<CTRL+C>
```

Приклад. Пересилання файлів.
Машина 1:
```
> nc -l 3000 > filename.out
```
Машина 2:
```
> nc host.example.com 3000 < filename.in
```

Приклад. Сервер:
```
# 6.sh
for i in 300{0..5}; { echo "Server" | ( nc -lknv "$i") & }

./6.sh
Connection from 127.0.0.1 port 3001 [tcp/*] accepted
Client 3001
Connection from 127.0.0.1 port 3004 [tcp/*] accepted
Client 3004
Connection from 127.0.0.1 port 3002 [tcp/*] accepted
Client 3002
Connection from 127.0.0.1 port 3003 [tcp/*] accepted
```

Приклад. Клієнт:
```
# 7.sh
for i in 300{0..5}; { echo "Client $i" | ( nc localhost "$i" ) }

./7.sh
Server
Server
Server
```

Приклад. Отримання домашньої сторінки із Web сервера:
```
$ echo -n "GET / HTTP/1.0\r\n\r\n" | nc pnu.edu.ua 80
```

Приклад. Відправлення e-mail на SMTP сервер:
```
> nc [-C] localhost 25 << EOF
HELO pnu.edu.ua
MAIL FROM:<user@pnu.edu.ua>
RCPT TO:<user2@pnu.edu.ua>
DATA
```

```
Body of email.
...
QUIT
EOF
```

Приклад. Перевірка стану портів у заданому діапазону (без встановлення з'єднання):

```
> nc -z pnu.edu.ua 20-30
Connection to host.example.com 22 port [tcp/ssh] succeeded!
Connection to host.example.com 25 port [tcp/smtp] succeeded!
```

Приклад. Отримання інформації про програмне забезпечення сервера:

```
$ echo "QUIT" | nc pnu.edu.ua 20-30
SSH-1.99-OpenSSH_3.6.1p2
Protocol mismatch.
220 host.example.com IMS SMTP Receiver Version 0.84 Ready
```

Приклад . Відкрити TCP з'єднання до 42 порту хосту `pnu.edu.ua`, використовуючи порт 31337 як джерело, з часом простою `timeout` 5 сек:

```
> nc -p 31337 -w 5 pnu.edu.ua 42
```

Приклад . Відкрити UDP з'єднання до порту 53 хосту `pnu.edu.ua`:

```
> nc -u pnu.edu.ua 53
```

Приклад. Відкрити TCP з'єднання до порту 42 pnu.edu.ua використовуючи 10.1.2.3 як IP для локального кінця з'єднання:

```
> nc -s 10.1.2.3 pnu.edu.ua 42
```

Приклад. Створити і прослуховувати сокет Unix домену:

```
> nc -lU /var/tmp/dsocket
```

Приклад. З'єднатися з портом 42 хосту `pnu.edu.ua` через HTTP проксі з IP-адресою 10.2.3.4, порт 8080:

```
> nc -x10.2.3.4:8080 -Xconnect pnu.edu.ua 42
```

Цей самий приклад, але з ідентифікацією користувача з username "`ruser`" як проксі:

```
> nc -x10.2.3.4:8080 -Xconnect -Pruser host.example.com 42
```


7. Команда `netstat`

Команда `netstat` відображає статистику активних підключень TCP, прослуховуваних портів комп'ютером, статистику Ethernet, таблиці маршрутизації, статистики IPv4 і IPv6.

Формат командного рядка:

```
netstat [-a] [-b] [-e] [-f] [-n] [-o] [-p протокол] [-r] [-s] [-t] [інтервал]
```

-a – відображення всіх підключень і очікуючих портів.

-b – відображення виконуваного файлу, який бере участь у створенні кожного підключення або очікує відповіді.

-e – відображення статистики Ethernet.

-f – відображення повного імені домену для зовнішніх адрес.

-n – відображення адрес і номерів портів у числовому форматі.
-o – відображення коду (ID) кожного процесу.
-p протокол – TCP, UDP.
-r – відображення вмісту таблиць маршрутів.
-s – відображення статистики протоколу.
-t – відображення поточного стану підключення.
-v – детальне виведення інформації при можливості.

Приклади.
Відобразити всі з'єднання у посторінковому режимі виведення на екран:

```
netstat –a | more
```

Відобразити всі з'єднання із статусом LISTENING, тобто порти які прослуховуються:

```
netstat –a | find /I ”LISTENING”
```

Відобразити всі з'єднання і зв'язані з ними програми:

```
netstat –ab
```

Приклад. Перевірка відкритих портів командою `netstat`:

```
> netstat -tulanp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Addres State PID/Program name
tcp 0  0 127.0.0.1:631  0.0.0.0:* LISTEN - tcp 0 0 127.0.0.1:25 0.0.0.0:*
LISTEN -
tcp 0  0       ::1:631       :::* LISTEN -
tcp 0  0       ::1:25        :::* LISTEN -
tcp 0  0        :::1716      :::* LISTEN 2104/kdeconnectd

> netstat pnu.edu.ua
Proto RefCnt Flags Type State  I-Node  Path
unix  2      [ ]   DGRAM      26897    /run/user/1000/systemd/notify
unix  3      [ ]   DGRAM       9260    /run/systemd/notify
unix  2      [ ]   DGRAM       9262    /run/systemd/cgroups-agent
```

8. Команда ss

Команда ss виводить статистику сокетів по TCP в більшому обсязі порівняно з іншими командами. Команда має наступні опції:
-n – не розкривати імена служб;
-r – не розкривати імена портів;
-a – показувати всі сокети;
-l – показувати слухаючі сокети;
-o – показувати інформацію таймера;
-e – показувати детальну інформацію про сонет;
-m – показати використання пам'яті сонетом;
-p – показати процес, який використовує сонет;
-i – показати внутрішню інформацію TCP;
-4 – показувати тільки сокети IP4;
-6 – показувати тільки сокети IP6;
-0 – показувати пакетні сокети;

-t – показувати тільки сокети TCP;
-u – показувати тільки сокети UDP;
-d – показувати тільки сокети DCCP;
-w – показувати тільки сокети RAW;
-x – показувати тільки сокети Unix.
Приклади.
`ss -t -a` – показати всі сокети TCP;
`ss -u -a` – показати всі сокети UDP.

9. Команда налаштування мережі ip

Синтаксис команди
`ip [опції] об'єкт команди [параметри]`

Опції:
`-v` – виведення інформації про команду.
`-s` – виведення статистичної інформації.
`-f` – протокол `bridge | dnet | inet | inet6 | ipx | link`
`-o` – виведення кожного запису з нового рядка.
`-4` – для –f inet.
`-6` – для –f inet6.
`-B` – для –f bridge.
`-O` – для –f link.

Об'єкти:
`address` – мережева адреса на пристрої.
`link` – фізичний мережевий пристрій.
`monitor` – монітор стану пристрою.
`neigh` – ARP.
`route` – керування маршрутизацією.
`rule` – правила маршрутизації.
`tunnel` – налаштування тунелювання.

Команди налаштування мережі:
`add, change, del, flush, get, list, show, monitor, replace, restore, save, set, update.`

Параметри (залежать від об'єкта і команди)
`dev ім'я пристрою` – мережевий пристрій.
`up` – включити.
`down` – виключити.
`llarrr` – MAC адреса.
`initcwnd` – розмір вікна перевантаження TCP при ініціалізації.
`window` – розмір вікна TCP.
`cwnd` – розмір вікна перевантаження TCP.
`type` – тип.

`via` – підключитися до роутера.
`default` – маршрут за замовчуванням.
`blackhole` – маршрут “чорна дірка”.
`prohibit` – маршрут “заборони”
`unreachable` – недосяжний маршрут.

Приклади використання команди `ip link`:
`ip link show` – показати стан всіх мережевих інтерфейсів
`ip link list up` - показати стан всіх включених інтерфейсів
`ip link set eth1 up/down` – включити/виключити `eth1`.

Приклади використання команди `ip neigbour`:
`ip neigh show` – показати всі записи ARP.

Приклади використання команди `ip address`:
`ip addres show` – показати сві ip адреси і їх інтерфейси
`ip addr add 1.1.1.13/24 dev eth0` – встановити ip адресу для інтерфейсу `eth0`
`ip addr del 1.1.1.13/24 dev eth0` – вилучити ip адресу для інтерфейсу `eth0`

Приклади використання команди `ip route`:
`ip r sh` - показати всі маршрути в таблиці маршрутизації.
`Ip route get 10.10.20.0/24 from 192.168.12.9` – відобразити маршрут до вказаної мережі від заданого інтерфейсу.

10. CGI-сценарій

CGI (Common Gateway Interface) сценарій це програма, яка виконується веб-сервером. CGI-сценарії розміщуються у спеціальному каталозі `/srv/www/`*`cgi-bin`*. CGI-сценарію дають розширення `.cgi`, щоб відрізнити його від сценаріїв загального призначення із розширеннями `.sh`.

CGI сценарій записує коротке HTTP повідомлення веб-серверу (заголовок) перед відправленням любої інформації. Найкоротший заголовок містить рядок з описанням різновиду даних, які буде повернено веб-переглядачу і кодування, а після нього новий порожній рядок.

```
printf ”Content-type: text/plain; charset=utf-8\n\n”
```

CGI-сценарій записується у каталог `/srv/www/`*`cgi-bin`* з правом на виконання і викликається із переглядача:

- у локальному режимі `http://localhost/cgi-bin/env.cgi;`
- у віддаленому режимі `http://192.44.152.137/cgi-bin/env.cgi`.

Взнати IP-адресу хоста можна командою `host`:

```
>  host pnu.edu.ua
192.44.152.137
```

Так як CGI-програма викликається на виконання переглядачем, то її не можна налагодити у консольному режимі. Найбільш загальні помилки, які виникають при виконанні CGI-програми

403 Forbidden — The script permissions are wrong or the script is not in a CGI directory
404 Not Found — The CGI script was missing or the wrong URL was used.
500 Internal Server Error—The script didn't return a proper CGI header.

Всі помилки, які записуються у стандартний потік помилок, записуються також і у журнал веб-сервера (log файл). Так як незручно у журналі веб-сервера шукати помилки, рекомендується захоплювати їх у сценарії і направляти у потік стандартного виведення на веб-сторінку. Для того щоб поверталася веб-сторінка, а не "плоский" (plain) текст, необхідно у вмісті сторінки вказати `text/html`:

```
printf "Content-type: text/html; charset=utf-8\n\n"
```

При виведенні помилок у веб-сторінку потрібно передбачити, щоб їх колір і шрифт відрізнявся.

10.1. Змінні середовища CGI-програми

CGI-програма має додаткові змінні середовища, які назначаються веб-сервером:

AUTH_TYPE — Authorization type if pages are password protected
CONTENT_LENGTH — Number of bytes being written to standard input (for POST forms)
CONTENT_TYPE — The form's content type
DOCUMENT_ROOT — The root directory of the Web server's document tree
GATEWAY_INTERFACE — The version of the CGI standard being used by the Web server
HTTP_ACCEPT — Types of data acceptable to the browser (for example, text/html)
HTTP_ACCEPT_CHARSET — Character set requested by the Web browser
HTTP_ACCEPT_ENCODING — Compression methods allowed by the Web browser (for example, gzip)
HTTP_ACCEPT_LANGUAGE — Language requested by the Web browser (for example,en for English)
HTTP_USER_AGENT — The browser used by the user
HTTP_HOST — The URL's hostname
HTTP_REFERER — The Web page executing this CGI program
PATH_INFO — Extra information included in the URL
PATH_TRANSLATED — PATH_INFO, as a file/directory under the root of the document tree
QUERY_STRING — For GET forms, the variables on the form
REMOTE_ADDR — IP of the user's computer
REMOTE_HOST — Hostname of the user's computer
REMOTE_USER — Username used when accessing password-protected pages
REQUEST_METHOD — Usually GET or POST
SCRIPT_NAME — Pathname of the script being executed
SCRIPT_FILENAME — The absolute pathname of the script being executed
SERVER_ADDR — IP address of the Web server
SERVER_ADMIN — Email address to email messages to the person in charge of the Web server
SERVER_NAME — Domain name
SERVER_PORT — The TCP/IP port used to connect to the Web server
SERVER_PROTOCOL — Version of HTTP used by the server
SERVER_SOFTWARE — Description of the Web server

10.2. Оброблення форм

HTML еквівалентом змінних середовища є форма. Кожна форма містить набір змінних, наприклад тег `<input type="hidden" name="user" value="Ivanenko">` є HTML еквівалентом `declare user="Ivanenko"`. Форма може містити багато інших тегів, атрибути яких

змінюються користувачем і відправляються CGI-програмі. Для відправлення даних використовується два методи GET і POST.

У методі GET змінні форми зберігаються у змінній середовища `QUERY_STRING` і передаються разом з URL адресою.

Для наступної html сторінки змінна середовища `QUERY_STRING` буде містити імена і значення двох змінних `user=Ivanenko&submit=Click+Me%21`

```
<html>
<head>
<title>Form Test</title>
</head>
<body>
<form action="http://localhost/cgi-bin/form.cgi>
<input type="hidden" name="user" value="Ivanenko">
<input type="submit" name="submit" value="Click Me!">
</form>
</body>
</html>
```

Одна з причин, чому важко обробляти форми, полягає у кодуванні інформації. Стандартне кодування форми `(x-www-form-urlencoded)` замінює знак "+" і не букво-цифрові символи їх шістнадцятковими ASCII кодами із знаком "%" попереду.

POST метод записує змінні у стандартний вхід, запобігаючи можливості переповнення буфера веб-сервера для довгого списку змінних. Змінні передаються окремо від URL адреси веб-сторінки. Коли сценарій читає змінні у циклі `while`, останній рядок не обробляється так як він містить тільки признак нового рядка.

```
while read LINE ; do
  echo "$LINE<br />"
done
  echo "$LINE<br />"
```

Таким чином виводиться та сама закодована інформація, яка міститься у змінній середовища `QUERY_STRING` для метода `GET`.

Запитання.

1. Команди роботи з мережею.
2. Стратегії клієнт-сервер.
3. IP адреса і сокет.
4. Синтаксис мережевих сокетів.
5. Робота Bash-клієнта із сонетами.
6. Створення сервера і клієнта з використанням команди `netcat`.
7. Команди `netstat, ss`.
8. Команда налаштування мережі `ip`.
9. Правила написання CGI-сценарію.
10. Оброблення форм методами GET і POST.

СПИСОК ЛІТЕРАТУРИ

Основний

1. Системне програмне забезпечення [Електронний ресурс] : конспект лекцій для здобувачів освітнього ступеня «бакалавр» зі спеціальності 123 Комп'ютерна інженерія освітньої програми Спеціалізовані комп'ютерні системи денної форми навчання / упоряд.: Є.Є. Федоров, Т.Ю. Уткіна ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси: ЧДТУ, 2023. – 106 с. – Частина 1. – режим доступу: https://elib.chdtu.edu.ua/e-books/5239
2. Методичні рекомендації до лабораторних робіт з дисципліни «Системне програмне забезпечення» для здобувачів освітнього ступеня «бакалавр» зі спеціальності 123 Комп'ютерна інженерія освітньої програми Спеціалізовані комп'ютерні системи денної форми навчання [Електронний ресурс] / упоряд.: Є.Є. Федоров, Т.Ю. Уткіна ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси: ЧДТУ, 2023. – 14 с. – Частина 1 – режим доступу: https://elib.chdtu.edu.ua/e-books/5238 .
3. Операційні системи [Електронний ресурс] : навчальний посібник / І. М. Федотова-Півень, І. В. Миронець, О. Б. Півень, С. В. Сисоєнко, Т. В. Миронюк ; за ред. В. М. Рудницького, Черкаський державний технологічний університет. – Харків : ТОВ «ДІСА ПЛЮС», 2019. – 216 с. – ISBN 978-617-7645-93-0.
4. Зайцев, В. Г. Операційні системи. / Навчальний посібник для студентів спеціальності 123 «Комп'ютерна інженерія» / В. Г. Зайцев, І. П. Дробязко. – Київ: КПІ ім. Ігоря Сікорського, 2019. – 240 с.
5. Навчально-методичний посібник до виконання курсової роботи з дисципліни "Системне програмне забезпечення" для студентів спеціальності 6. 050102 "Комп'ютерна інженерія" всіх форм навчання [Електронний ресурс] / уклад. : І. М. Федотова-Півень, О. Б. Півень ; М-во освіти і науки України, Черкас. держ. технол. ун-т. – Черкаси : ЧДТУ, 2015. – 60 с.
6. Системне програмне забезпечення: навчальний посібник з дисципліни «Системне програмне забезпечення» для студентів базового напряму 6.050102 «Комп'ютерна інженерія»/ Укл.: І. В. Мороз, Л. О. Березко, О. Ю. Бочкарьов. – Львів: Видавництво Національного університету «Львівська політехніка», 2014. – 162 с.
7. Лабораторний практикум з дисципліни "Системне програмне забезпечення" для студентів напряму підготовки 6.050903 "Телекомунікації" всіх форм навчання [Електронний ресурс] / уклад. А. В. Чепинога. – Черкаси: ЧДТУ, 2010. – 66 с.

Додатковий

8. Nick Aleks, Dolev Farhi. Black Hat Bash: Creative Scripting for Hackers and Pentesters. No Starch Press, 2024. – 344 p.
9. Michael Kofler. Scripting: Automation with Bash, PowerShell, and Python—Automate Everyday IT Tasks from Backups to Web Scraping in Just a Few Lines of Code. Rheinwerk Computing, 2024. – 500 p.
10. Richard Blum, Christine Bresnahan. Linux Command Line and Shell Scripting Bible 4th Edition. Wiley, 2021. – 832 p.
11. Paul Troncone, Carl Albing.Cybersecurity Ops with bash: Attack, Defend, and Analyze from the Command Line 1st Edition. O'Reilly Media, 2019. 303 p.

12. William Shotts. The Linux Command Line, 2nd Edition: A Complete Introduction. No Starch Press, 2019. – 504 p.
13. Carl Albing. bash Cookbook: Solutions and Examples for bash. 2nd Edition. O'Reilly Media, 2017. – 723 p.

Інформаційні ресурси

14. Linux Mint 22 [електронний ресурс]: – Режим доступу: https://linuxmint.com/ – назва з екрану
15. Canonical Ubuntu. [електронний ресурс]: – Режим доступу: https://ubuntu.com/ – назва з екрану
16. Open Suse [електронний ресурс]: – Режим доступу: https://www.opensuse.org/ – назва з екрану
17. GNU BASH: – Режим доступу: https://www.gnu.org/software/bash/ – назва з екрану.